A Knowledge-Driven Approach for Dynamic Reconfiguration of Control Design in Internet of Things and Cyber–Physical Systems

Amar Banerjee and Venkatesh Choppella

***Abstract*—Dynamic control software reconfiguration for the Internet of Things (IoT) and cyber–physical systems (CPSs) is crucial for adaptable and efficient automation. This article presents a knowledge-driven architecture enabling dynamic device reconfiguration using the Web ontology language (OWL) and terse triple language (TTL) formats. Key components include a capability ontology, session-type information for sequencing and concurrent operations, and an integrated development environment (IDE) for automated control design. The capability ontology standardizes machine capabilities, facilitating device integration based on their capabilities, while session-type information ensures correct sequencing and synchronization of machine functions. The IDE platform supports dynamic reconfiguration by automating device selection, control strategy formulation, and system adjustments across diverse use cases. The architecture has been validated in real-world scenarios, including smart meeting rooms, warehouse automation, and energy management, showing a reduction in manual configuration time (up to 50%), development time (86% in some cases), and error rates (30%). Benchmarking results indicate faster code generation (40% improvement) and efficient component integration across different CPS environments. Challenges like computational complexity, scalability, and integration with existing systems highlight limitations. Future research will explore further optimizations and broader applicability to ensure low-latency, high-accuracy, and seamless integration in complex CPS. This work advances dynamic control software reconfiguration by providing a flexible solution that enhances CPS reliability and efficiency through a knowledge-driven approach.**

***Index Terms*—Capability ontology, control design, control software, cyber–physical systems (CPSs), dynamic reconfiguration, industrial automation systems, Internet of Things (IoT), knowledge-driven approach (KDA).**

I. Introduction

IN CYBER–PHYSICAL systems (CPSs) and the Internet of Things (IoT), control software coordinates interactions between physical and virtual entities, enabling applications, such as industrial automation, robotics, transportation, energy management, and intelligent cities [1]. This control must be dynamic, adaptive, and responsive to evolving system conditions and environmental factors. Furthermore, it must ensure low latency and high accuracy in real-time CPS environments to maintain performance and reliability. Achieving this adaptability is central to dynamic reconfiguration [2].

Control software is essential in modern industrial systems, managing real-time interactions between sensors, actuators, machines, and robots. It ensures efficient, safe operation across manufacturing, healthcare, aerospace, and energy [3]. The significance of control software has increased with the advent of *Industry 4.0* [4], where the integration of CPS, robotics, and IoT has enabled automation, predictive maintenance, and optimization in modern factories, healthcare, and aerospace [5]. For instance, it controls hundreds of machines in real time, ensures precision in robotic surgeries, and manages safety-critical systems like autopilot in aircraft. The Ethiopian Airlines Flight 302 crash in 2019, where control system malfunctions contributed to the loss of 157 lives, underscores the need for dependable control software in safety-critical domains [6]. This incident underscores the need for dependable, validated control software, particularly in safety-critical domains.

Dynamic reconfiguration involves adjusting a system's structure, functionality, or performance characteristics in response to changing conditions to maintain consistent performance and reliability [7]. It can be compared to a conductor modifying an orchestra's performance in real time to ensure harmony.

For example, in a smart city, interconnected systems, such as traffic management, power grids, public transportation, and emergency services, must adapt quickly to events like power outages or traffic incidents. This may involve rerouting traffic, adjusting transportation schedules, notifying emergency services, and redistributing power, demonstrating dynamic reconfiguration in action [8].

A. Problem of Reconfiguration

Reconfiguration is necessary to handle changes, such as hardware upgrades, process modifications, or unexpected device failures. However, it presents challenges, including the need for real-time responses, managing the complexity of industrial systems, and ensuring uninterrupted operation while maintaining safety. Additionally, incorporating domain-specific knowledge and managing device-specific protocols further complicates the process [9].

Received 19 August 2024; revised 20 October 2024; accepted 24 October 2024. Date of publication 29 October 2024; date of current version 21 February 2025. *(Corresponding author: Amar Banerjee.)*

The authors are with the Software Engineering Research Center, International Institute of Information Technology Hyderabad, Hyderabad 500032, India (e-mail: amar.banerjee@research.iiit.ac.in; venkatesh.choppella@iiit.ac.in).

Digital Object Identifier 10.1109/JIOT.2024.3487578

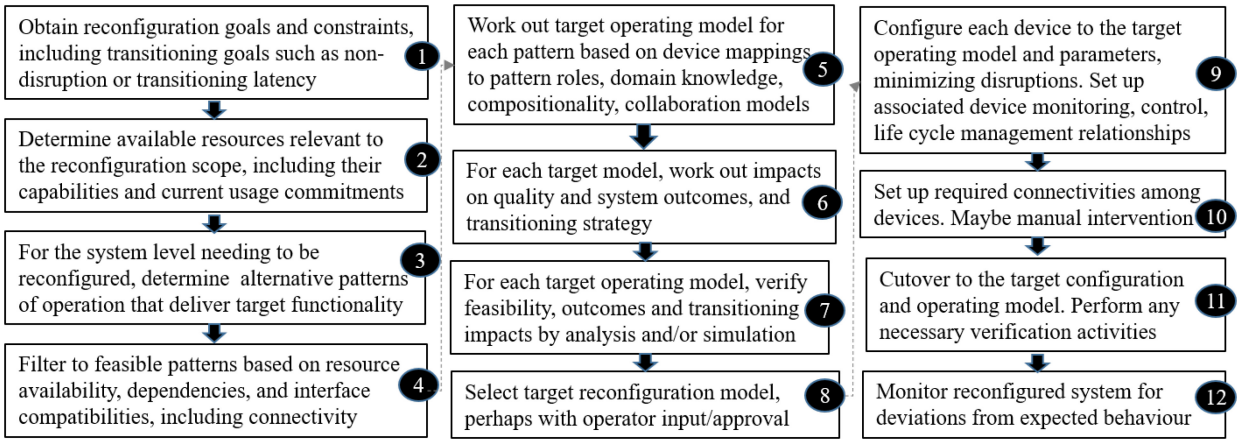


Fig. 1. System reconfiguration life cycle.

B. Reconfiguration Scenarios

Reconfiguration scenarios can be broadly classified as follows.

1) *Device Upgrade:* When a device is upgraded, the control software must adapt to the new device's communication protocol. For instance, replacing a robotic arm in a manufacturing system with a newer model requires reconfiguring the software to support the updated protocol [9].
2) *Process Change:* Changes in processes necessitate reordering device interactions. In an assembly line, a new requirement may involve inspecting parts with a camera before assembly. The control software must be reconfigured to ensure the camera inspection occurs first [9].
3) *Device Failure:* In case of device failure, the control software must bypass the faulty component to maintain operations. For example, if a conveyor belt fails, the robotic arm may need to retrieve parts directly from a temporary storage area [9].

Dynamic reconfiguration requires addressing challenges, such as understanding device capabilities, behaviors and resources, ensuring reliability, optimizing real-time performance, and minimizing disruptions. Overcoming these challenges demands deep domain knowledge and a thorough understanding of devices, their interactions, and related tasks.

Device knowledge [10], encompassing device capabilities, behaviors, and operational conditions, is crucial for dynamic reconfiguration. Capturing and using this knowledge provides insights into how devices adapt to new conditions or requirements.

Involving domain experts in control software development is essential for capturing and utilizing domain knowledge. However, effective communication between domain experts and software developers is often challenging due to differing expertise and technical backgrounds. Thus, it is vital to ensure that domain knowledge is comprehensive, up-to-date, and accurately integrated into control software to meet system requirements [11].

C. Reconfiguration Life Cycle

The reconfiguration life cycle involves a sequence of tasks triggered by events, such as sensor failure, new device introduction, or adapting to constraints. Dynamic reconfiguration automates the analysis, identification, and verification of alternative configurations to minimize disruptions and maintain high reliability.

Supporting the reconfiguration life cycle requires a capability ontology that provides comprehensive information about domain workflows, context, goals, and system impacts. Fig. 1 illustrates the system-level reconfiguration process.

D. Scope of the Work

This work focuses on a knowledge-driven approach (KDA) for dynamic reconfiguration of control software in CPS and IoT, emphasizing a capability ontology-based architecture for synchronization and real-time reconfiguration. While factors, such as data processing capabilities, latency, and accuracy, are acknowledged as necessary, they are not the main focus of this article. The primary focus is control design reconfiguration, providing a framework for integrating and adapting control software in dynamic environments. Future work will explore extending the framework to address real-time data processing challenges.

The proposed architecture utilizes a KDA supported by a capability ontology that captures and leverages knowledge about machines, their behaviors, capabilities, and sessions. Ontologies, such as Web ontology language (OWL) and Turtle (TTL) formats, offer a structured representation of this knowledge. By focusing on a capability-based approach, the architecture enables effective synchronization and execution of machine functions, addressing the complexities of integrating multiple devices in control software.

The structure of this article is as follows. Section II reviews related work on dynamic reconfiguration. Section III presents the proposed approach, and Section IV formally defines the key concepts. Section V describes the capability ontology, while Section VI details the composition architecture. Section VII discusses the composition algorithm. Section VIII

TABLE I
COMPARATIVE ANALYSIS OF RELATED WORKS

Approach	Contribution	Limitations	Proposed Approach Advantage
Fadhlillah et al. [7] (2022)	Variability management in CPPSs	Focuses on contextual variations	Emphasizes dynamic reconfiguration
Thuluva et al. [12] (2020)	Extends Node-RED for IoT with semantics	Limited evaluation; Lacks tool assessment	Comprehensive tool evaluation
C4I [13] (2020)	Semantic Web for manufacturing capabilities	Formal logic; Overlooks uncertainties	Dynamic updates; Handles uncertainties
Sari et al. [14] (2020)	Ontology for runtime binding of IoT device services	Limited to basic device service relationships	Supports comprehensive reconfiguration, including task scheduling
Nguyen-Anh et al. [15] (2019)	Bayesian and ontology-based reconfiguration for IoT	Complexity in Bayesian network setup; Handles specific cases	Unified framework with session-type reasoning for diverse scenarios
Wan et al. [16] (2018)	Ontology-based resource reconfiguration for MCPS	Static modeling; Lacks real-time adaptation	Incorporates dynamic reconfiguration; Session-type management
MaRCO [17] (2019)	Standardizes manufacturing resource capabilities	Static descriptions; No real-time adaptability	Real-time adaptability; Detailed behaviors
Pentga et al. [18] (2016)	CPS decision support framework	Ignores non-machine entities	Holistic system view; Includes non-machine entities
Alsafi et al. [19] (2010)	Ontology-based reconfiguration for mechatronics	Limited flexibility; Complex scenarios	Enhanced flexibility; Session-type integration

covers implementation within an integrated development environment (IDE), followed by three use cases in Section IX. Section X presents the evaluation and Section XI outlines limitations. Finally, Section XII concludes this article.

II. RELATED WORK

Pursuing dynamic reconfiguration in control software for cyber–physical systems (CPSs) remains challenging, as it involves adapting to the evolving needs of systems and environments. This section reviews state-of-the-art approaches, highlighting gaps and limitations. Table I summarizes these approaches.

Fadhlillah et al. [7] proposed a multidisciplinary delta-oriented variability management approach for CPPSs, focusing on managing variability. However, it emphasizes contextual variations rather than machine reconfiguration and does not fully account for various domain processes in plants.

Weser et al. [13] developed the Capability For Industry (C4I) model using Semantic Web technologies to formalize manufacturing capabilities. Although it facilitates knowledge representation and extension, its reliance on formal logic and OWL may not fully capture the complexities of real-time production environments, potentially overlooking uncertainties and tacit knowledge. Merdan et al. [20] offered a model-driven engineering approach for CPS configuration based on knowledge representation, but it lacks a comprehensive solution for dynamic reconfiguration and system synchronization.

Sari et al. [14] introduced an ontology for the dynamic binding of IoT applications at runtime. While it enhances device service descriptions and supports runtime binding, it does not extend to system-level reconfiguration, particularly for complex workflows and task sequencing.

Nguyen-Anh and Le-Trung [15] developed a reconfiguration framework combining ontology and Bayesian networks to manage IoT device contexts. Although Bayesian inference offers probabilistic reasoning in uncertain environments, the complexity of setting up Bayesian networks may limit its applicability to diverse CPS scenarios, and it does not incorporate formal session-type reasoning needed for concurrent operations in complex systems.

Järvenpää et al. [17] introduced the manufacturing resource capability ontology (MaRCO) to standardize manufacturing resource capabilities. While it supports automated decision-making in multivendor environments, its focus on static resource descriptions limits its effectiveness in dynamic reconfiguration, as it cannot account for real-time changes in resource conditions or detailed behavioral nuances needed in dynamic settings.

Alsafi and Vyatkin [19] proposed an ontology-based reconfiguration agent for intelligent mechatronic systems. Despite its innovative approach, the reliance on OWL and the MASON ontology limits flexibility in handling complex reconfiguration scenarios. The method does not adequately address rapid changes in resource capabilities, production requirements, or resource performance variability, which could impact production tasks.

Wan et al. [16] proposed an ontology-based method for reconfiguring manufacturing CPSs using a multilayer architecture for device knowledge integration. Although the framework offers structured modeling and reasoning, it lacks support for real-time dynamic reconfiguration and relies on static resource descriptions, limiting its capability for frequent updates.

Pentga and Austin [18] presented a knowledge and reasoning framework for CPS decision support, focusing on developing knowledge structures for correct-by-design CPS models. While effective for semantic parameter handling and decision-making, it does not account for machine variability and the role of nonmachine entities, such as human interventions, which are crucial for realistic plant operations.

In summary, existing approaches provide valuable contributions to knowledge-driven development and variability management, but a gap remains in the comprehensive synchronization and dynamic reconfiguration of machines in CPS. Current methods lack a formal approach for capturing, representing, and utilizing domain knowledge, interaction protocols, and tasks necessary for adaptive control software and controller synthesis.

This analysis highlights the need for an approach that standardizes capturing, representing, and using knowledge about machines, enabling dynamic reconfiguration. The following

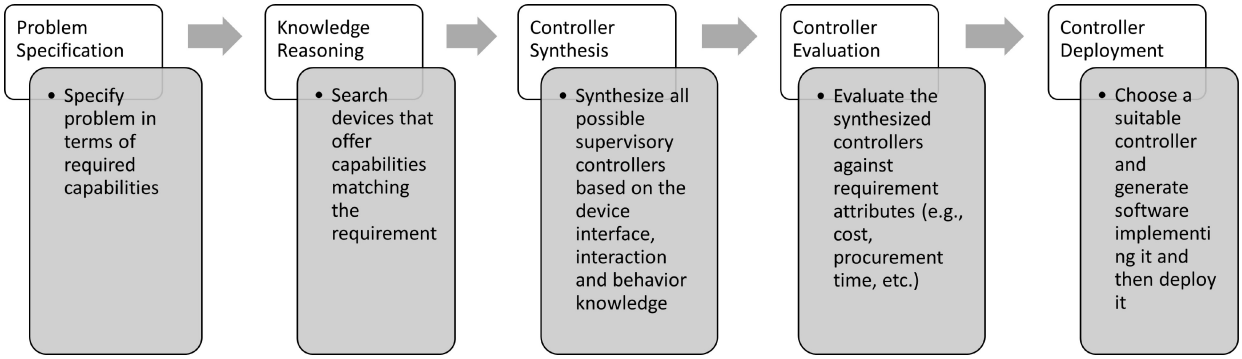


Fig. 2. Controller synthesis for dynamic reconfiguration.

sections present our proposed knowledge-driven architecture, addressing these gaps and providing solutions for control software development in CPS environments.

III. Capability-Driven Approach for Controller Synthesis

Adapting to changes in the environment or system state is crucial for maintaining optimal performance and functionality [21]. This necessitates an approach where the inherent capabilities of a device or system are leveraged to design effective control strategies at both design and runtime. By capturing and understanding these capabilities, dynamic reconfiguration becomes feasible, enabling the system to adapt and respond to changes effectively. This forms the foundation of our proposed solution, addressing the challenges of dynamic reconfiguration in CPS by utilizing capability knowledge to automate controller synthesis.

The capability-driven approach for controller synthesis utilizes comprehensive knowledge of device capabilities, session types, and workflows to design an effective controller for a plant system [22]. This approach integrates knowledge representation, reasoning, and synthesis techniques to automate the controller design process, which can be triggered dynamically. Fig. 2 illustrates the high-level view of the controller synthesis process that supports dynamic reconfiguration using capability knowledge.

A key novelty of this approach is integrating capability knowledge with session-type information, enabling precise sequencing and concurrent execution of machine functions. This integration ensures the control system can dynamically reconfigure in response to changing operational conditions, minimizing disruptions and maintaining system reliability.

The KDA for controller synthesis involves several steps.

1) *Knowledge Acquisition:* Gather knowledge from devices involved in the control system, including capabilities, specifications, and operational characteristics.
2) *Knowledge Representation:* Structure the acquired knowledge into a format, such as an ontology, to facilitate reasoning and manipulation.
3) *Problem Specification:* Define the specific control problem in the plant system, forming the basis for controller synthesis.
4) *Knowledge Reasoning:* Apply reasoning techniques to infer relationships and dependencies between devices, their capabilities, and the control problem.
5) *Controller Synthesis:* Generate a set of candidate controllers to address the control problem, using algorithms and optimization techniques based on the available knowledge.
6) *Controller Evaluation and Selection:* Assess the synthesized candidate controllers against predefined criteria and performance metrics, selecting the optimal controller.
7) *Controller Deployment:* Deploy the final controller to the plant system, integrating it with the devices and sensors.

In control software, device capabilities refer to specific functions or features a component can perform [10]. These capabilities capture the device's potential to affect changes in its environment and state, accompanied by specific behaviors and characteristics. By leveraging these capabilities in the synthesis process, the proposed approach ensures that the control system remains flexible and adaptable across various operational conditions.

A. Device Capabilities

Device capabilities in control software refer to the specific functions or features a component can perform [10]. These capabilities define the device's ability to affect changes in its environment and, potentially, its state, with particular behaviors and characteristics. For example, a device may have a "toasting" capability for altering the state of bread or a "projection" capability to display images.

Capturing device capabilities is essential for compositional reasoning, configuration management, and controller synthesis. Capability descriptions typically involve domain-specific terminology and define validity constraints over a system's sequence of observables, including timing and reliability [23].

The capability-driven approach is designed to accommodate diverse types of CPS, such as those in industrial automation, smart grids, and autonomous vehicles. The approach can handle varying conditions, including network latency and concurrent operations, ensuring the control system remains effective across different domains by capturing and utilizing detailed capability knowledge.

B. Capabilities, Interfaces, Interactions, and Context

Capabilities, interfaces, interactions, and context are crucial for device self-descriptions. A capability-based approach requires comprehensive descriptions of these aspects to enable effective composition, reasoning, and lifecycle activities. Key components of a capability self-description include the following.

1) *Capabilities:* Detailed descriptions of the device's functionality, features, and quality characteristics. These descriptions should be standardized to enable dynamic composition across devices from multiple providers [10].
2) *Interfaces:* Specifications of all layers of interfaces, including physical, technological, technical, and functional interfaces. This includes operational, control, and lifecycle management interfaces [11].
3) *Interactions:* Usage protocols associated with each device interface, facilitating compositional reasoning. This includes session types for interaction, participant roles, resulting state changes, exhibited behavior, and quality characteristics [24].
4) *Context:* Dependencies and assumptions about resources, environment, and contextual entities involved in the device's operation. This information enables reasoning about the well-formedness of a target configuration.
5) *Device Configurability:* Specifications of operating modes, configuration parameters, and their impact on device functions, interactions, and quality characteristics [23].
6) *Variations, Failure Modes, and Handling:* Descriptions of variations in inputs and device states, including undesired variations like security and safety threats and failure modes. This also includes actions for handling fault situations [24].

This approach also addresses scalability and computational complexity challenges by optimizing the synthesis process to ensure low latency and high accuracy in real-time scenarios. Integrating semantic rules and capability knowledge further enhances the effectiveness of controller synthesis, even in complex CPS environments.

Domain and contextual knowledge specifications are also necessary for capability composition and reasoning, encompassing goal decomposition knowledge, process specifications, stakeholder value specifications, and operational environment specifications.

C. Capabilities Versus Services Versus Functionalities Versus Features

To clarify the focus of the capability-driven approach, it is essential to distinguish between capabilities, services, functionalities, and features. While services and functionalities refer to broader aspects of software applications, capabilities represent the specific intrinsic abilities of a device or system that enable it to perform particular tasks. This distinction is crucial for designing controllers effectively leveraging device capabilities in dynamic reconfiguration scenarios.

In summary, the capability-driven approach for controller synthesis offers a flexible and scalable solution for dynamic reconfiguration in CPS. By focusing on the intrinsic capabilities of devices and integrating session-type information, the approach ensures that control systems can adapt to varying operational conditions while maintaining high performance and reliability. The next section provides formal definitions and a comprehensive representation of capabilities, which are the foundation for the proposed knowledge-driven architecture.

IV. Formal Framework

We begin by defining a system of interest or a plant that exhibits specific behavior.

Definition 1 (System of Interest/Plant): A System of Interest or Plant is a transition system [25] S defined by a six-tuple

$$S := \left(X_S, X_S^0, U_S, \underset{S}{\rightarrow}, Y_S, h_S\right) \tag{1}$$

where

1) X_S is the state space of S, and X_S^0 is the set of initial states.
2) U_S is the set of inputs or actions for S.
3) $\underset{S}{\rightarrow} \subseteq X_S \times U_S \times X_S$ is the transition relation of S, which describes how the system transitions from one state to another given an input.
4) Y_S is the set of observable outputs.
5) $h_S{:}X_S \rightarrow Y_S$ is the system's view function that maps its state to its observable output.

This formalization is crucial for modeling the dynamic behavior of CPS, where adaptability and responsiveness to changes are key. The transition system framework ensures that the system's behavior can be predicted and controlled, facilitating effective dynamic reconfiguration. We assume that all systems are non-blocking, meaning that for every state x and input u, at least one state x' exists such that $x \xrightarrow[S]{u} x'$.

The system of interest, or the plant, is the entity we want to control or manage. In the context of capabilities, each state of the system can be seen as a manifestation of a specific capability. The transition system formalism allows us to model the system's dynamic behavior, which is crucial for understanding how the system can be reconfigured to adapt to changes in the environment or requirements.

The formal framework introduced here provides the mathematical foundation for the capability-driven approach. By defining systems as transition systems, we ensure the control software can manage and adapt to various states and inputs, enabling robust dynamic reconfiguration across different CPS domains.

Definition 2 (Runs, Trajectories, Traces, and Behavior): Runs, traces, and behavior are defined as follows: A run in S is a sequence

$$\left\{x_i \xrightarrow[S]{u_i} x_{i+1}, i \in \mathbb{N}\right\}. \tag{2}$$

TABLE II
COMPARISON OF CAPABILITIES, SERVICES, FUNCTIONALITIES, AND FEATURES

Capabilities	Functionalities	Features	Services
Specific functions or features that a device or system is capable of performing.	The overall range of software application features, capabilities, and behaviors.	Individual tools, functions, or software components that perform a specific task or set of tasks.	A specific set of functions or capabilities a software or company provides to accomplish a particular task or meet a specific need.
Capture the intrinsic abilities and potential of the device or system.	Encompass all specific capabilities of the software, as well as other aspects such as the user interface, usability, and performance.	Typically accessed through menus, buttons, or other interface elements.	Often accessed through an API or other web-based interface.
Highlight the strengths or unique selling points of a device or system.	Describe the overall purpose and usefulness of the software as a whole.	Vary in complexity, from simple tasks such as changing the font size to more advanced tasks like 3D modelling.	Can range from simple services like email or messaging to more complex services like machine learning or data analysis.

A sequence $\{x_i\}$ is a *trajectory* if there exists a run $\{x_i \xrightarrow{u_i} Sx_{i+1}\}$ in S. A sequence $\{y_i\}$ is a *trace* in S if there exists a trajectory $\{x_i\}$ such that $y_i = x_i$.

1) The trace of a run x is the sequence y such that $y[i] = h_S(x[i])$.
2) X^ω and U^ω represent sequences of states and inputs, respectively, and $\mathbb{N}$ represents the set of natural numbers.
3) The set of all runs originating in a set A is denoted by $\mathcal{R}_A$.
4) The set of all traces of runs in S originating in a set $s \subseteq X$ is denoted by $\mathcal{B}_S(s)$.
5) The behavior of S is denoted by $\mathcal{B}_S(X^0)$.

This formalism of runs, traces, and behavior is essential for understanding how a CPS responds to various inputs and conditions over time. It allows for the precise modeling of system behavior, which is critical for ensuring that the control software can adapt to dynamic changes while maintaining performance and reliability.

Example 1: Let us consider an example of a warehouse automation scenario.

In this scenario, we have a warehouse with shelves labeled Shelf A and Shelf B and a bin where objects are supposed to be placed. The automation system consists of a robot responsible for picking up objects from the shelves and putting them in the bin.

For example, at time step 0, the system's initial state is shown in the table. The robot is located at Shelf A, and the object is initially placed on Shelf A. At each subsequent time step, the system's state evolves, representing the movement and interaction between the robot, the shelves, and the object. The trace demonstrates the observed states of the system at each time step.

This is a simplified example for illustrative purposes, and real-world warehouse automation systems would involve additional complexities and considerations.

The warehouse automation example illustrates how the formal framework can be applied to real-world scenarios. By modeling the system's runs and traces, we can predict and control the system's behavior, ensuring that dynamic reconfiguration processes are effective and reliable in complex environments.

Definition 3 (Problem/Requirement): A **problem** P is defined as a required change in observables of a Plant from an initial observable y to a final observable y'. It can be represented as a function

$$R : Y \to Y \tag{3}$$

where Y represents the set of all possible system observables.

Example 2: In the warehouse example, we can define a procurement problem of picking an item from an initial location and placing it in a final location. The initial and final observables can be represented as follows:

$$X_i^P = (< x_1, y_1, z_1 >, < x_2, y_2, z_2 >) \tag{a}$$

$$X_f^P = (< x_3, y_3, z_3 >, < x_3, y_3, z_3 >) \tag{b}$$

(assuming the item and the robot would have the same location)

$$\text{Procurement problem } P_p : (< x_1, y_1, z_1 >, < x_2, y_2, z_2 >) \underset{\delta t}{\to} (< x_3, y_3, z_3 >, < x_3, y_3, z_3 >) \tag{c}$$

(assuming there are no known intermediary states)

The problem/requirement definition is central to the dynamic reconfiguration process, as it specifies the desired outcome that the system must achieve. By formalizing problems this way, the framework ensures that the control software can dynamically adjust to meet these requirements, even as conditions change. The problem or requirement is what we want the system to achieve. This directly relates to the system's capabilities, as the system needs the necessary abilities to solve the problem. The problem or requirement for dynamic reconfiguration can change over time, necessitating system configuration changes.

Definition 4 (Problem Solving): A problem P is considered solved when the current observable of the plant matches the required final observable:

$$Y_S = Y_f \implies \text{solved}(P) = \text{True}. \tag{4}$$

Y_S represents the current observable of the system, and Y_f represents the final observable.

In the warehouse example, if after a specific time interval t', the location of the robot and the item is set to $(< x_3, y_3, z_3 > , < x_3, y_3, z_3 >)$, then the current state of the environment $X_{EoI} = (< x_3, y_3, z_3 >, < x_3, y_3, z_3 >)$ matches the required final state of the problem P_p.

The formal problem-solving framework ensures that the control system can verify when a problem has been resolved. This capability is critical for maintaining system reliability and ensuring that dynamic reconfiguration processes achieve the intended outcomes. Problem-solving is changing the system's state to meet a requirement. This is directly related to the system's capabilities, as the system needs to exercise the right capabilities to solve the problem. Regarding dynamic reconfiguration, problem-solving can involve changing the system's configuration to adapt to new issues or requirements.

Definition 5 (Session Type): Let T be a set of session types [26], $\mathcal{P}$ be a set of participant names and M be a set of message types.

The following grammar defines the session types:

$$T ::= \text{end} \mid P!M.T \mid P?M.T. \tag{5}$$

Here

1) "end" denotes the termination of a session.
2) $\mathcal{P}!M.T$ denotes a session where participant $\mathcal{P}$ sends a message of type M and then continues as T.
3) $\mathcal{P}?M.T$ denotes a session where participant $\mathcal{P}$ receives a message of type M and then continues as T.

Session types are crucial for ensuring that the interactions between different components of the CPS are well-structured and predictable. By formalizing session types, the framework supports synchronizing and coordinating various devices, which is essential for dynamic reconfiguration in complex environments.

Definition 6 (Capability): A **capability** F_P for a problem P is defined as a property of a machine S such that when S interacts with one or more entities E using a session type T, the problem P is solved. It is denoted as $M \models F_P$, where $\models$ represents "satisfies":

$$T : S \leftrightarrow E \xrightarrow{\text{results}} \text{solves}(P) = \text{True} \implies M \models F_P. \tag{6}$$

$M \models F_P$ means that machine M has the capability F_P.

The formalization of capabilities is a key component of the framework, allowing the control software to identify and leverage the specific abilities of each device. This capability-driven approach ensures that the system can dynamically reconfigure itself to meet new requirements or adapt to changes in the environment. The capability of a system refers to the set of requirements it can meet. Each capability has a protocol consisting of interconnects, initialization, and a sequence of interactions necessary with the environment to implement a requirement.

A capability allows a system to solve a specific problem when interacting with other entities. This is the core concept that we are focusing on. For dynamic reconfiguration, the system's capabilities determine possible configurations and how the system can be adapted to meet new requirements.

Definition 7: **System composition** is defined as: Let A and B be two systems connected via an interconnect $\mathcal{I}$. The *composition of* A and B, denoted $A \times_{\mathcal{I}} B$ is a system C defined as follows:

$$X_C = \mathcal{I}|_{X_A \times X_B} \tag{a}$$
$$X_C^0 = X_C \cap X_A^0 \times X_B^0 \tag{b}$$
$$U_C = U_A \times U_B \tag{c}$$
$$Y_C = Y_A \times Y_B \tag{d}$$
$$h_C(x_A, x_B) = (h_A(x_A), h_B(x_B)) \tag{e}$$
$$(x_A, x_B) \longrightarrow u_{CC} \left(x_A', x_B'\right). \tag{f}$$

Iff

1) $x_A \longrightarrow u_A A x_A'$,
2) $x_B \longrightarrow u_B B x_B'$, and
3) $(x_A, x_B, u_A, u_B) \in \mathcal{I}$.

.

A transition in the composite system is obtained by constraining the transitions of the individual component subsystems so that they respect the interconnect.

System composition formalizes integrating multiple systems into a cohesive whole, enabling the dynamic reconfiguration of CPS. By defining the interactions and interconnects between systems, the framework supports the seamless integration of new components and the adaptation of the control software to changing requirements.

The problem of reconfiguration can be viewed as a problem of interconnects between different systems. An interconnect between two systems can be seen as a relation that constrains their possible interactions. Given systems A and B connected via an interconnect $\mathcal{I}$, the composition of these systems, denoted as $A \times_{\mathcal{I}} B$, is a system that constrains the transitions of the individual component subsystems so that they respect the interconnect.

In the context of dynamic reconfiguration, system composition is particularly important as it allows the control software to seamlessly integrate new devices and adapt to new configurations without disrupting the overall system functionality. By formalizing the composition process, the framework ensures that reconfiguration can be achieved efficiently and reliably, even in complex CPS environments.

Reconfiguration involves modifying the composition and interconnectivity of a system to meet new requirements. The synthesis of solutions involves connecting multiple components to implement a (more substantial) requirement. This may include introducing an additional element, such as a controller, and specifying the interconnects.

The synthesis of solutions within this formal framework is guided by the need to achieve a well-formed, reliable configuration that meets the specified requirements. By defining clear protocols for composition and reconfiguration, the framework ensures that the control software can effectively adapt to changing conditions, maintaining high performance and reliability in real-time scenarios.

A. Example: Warehouse Automation Scenario

We illustrate their application using a warehouse automation scenario to provide a practical understanding of the formal definitions introduced. This example demonstrates how to transition systems, session types, and system composition are utilized to model and manage the dynamic behaviour of a cyber-physical system. In this scenario, a robot is responsible for transporting objects from various shelves to a designated

TABLE III
TRACE OF THE WAREHOUSE AUTOMATION SYSTEM

Time Step	Robot Location	Object Location
0	Shelf A	Shelf A
1	Shelf B	Shelf A
2	Shelf B	Shelf A
3	Shelf B	Robot Gripper
4	Bin	Robot Gripper
5	Bin	Bin

bin in a warehouse. The automation system must adapt dynamically to changes in object locations, robot tasks, and environmental conditions.

1) Modeling the System as Transition System: We model the warehouse automation system as a transition system S

The trace in Table III illustrates the observed sequence of states in the system. At each time step, the system's state changes according to the robot's actions, demonstrating the transition system's capability to model dynamic behavior.

2) Session Types for Coordination: The system employs session types to formalize communication protocols between the robot and other entities (e.g., a central controller). For example:

$$T ::= \text{Controller!MoveTo(Shelf B)}.T \mid \text{Robot?ObjectPicked}.T. \tag{7}$$

Here

1) Controller!MoveTo(Shelf B).T indicates that the controller sends a "MoveTo(Shelf B)" command to the robot.
2) Robot?ObjectPicked.T indicates that the robot acknowledges the "ObjectPicked" message before continuing.

Session types ensure that interactions follow a well-defined sequence, reducing the risk of miscommunication during reconfiguration.

3) Dynamic Reconfiguration and System Composition: The reconfiguration problem involves adapting the system to handle new tasks or respond to environmental changes. For instance, the system must reconfigure its control sequence if the robot needs to retrieve objects from a new location (e.g., Shelf C).

The system composition for integrating a new shelf is represented as

$$S_{\text{new}} = S_{\text{current}} \times_{\mathcal{I}} S_{\text{Shelf C}} \tag{8}$$

where $S_{\text{Shelf C}}$ represents the transition system for the new shelf, and $\mathcal{I}$ defines the interconnect specifying how $S_{\text{Shelf C}}$ is integrated with the existing system. The reconfiguration ensures that the robot can transition to states involving Shelf C while preserving the overall system behavior.

4) Problem Solving in the Warehouse Scenario: A problem P may involve picking up an object from a specific location and placing it in the bin. The system solves the problem by transitioning from the initial state to the final state where the object is in the bin

$$\text{Solve}(P) = (\text{Robot Location} = \text{Bin}) \wedge (\text{Object Location} = \text{Bin}). \tag{9}$$

The robot's capabilities (e.g., moving, picking up objects) and session-type interactions (e.g., acknowledging task completion) ensure the problem-solving process is dynamically adjusted based on changing task requirements.

The warehouse automation scenario illustrates the application of formal definitions such as transition systems, session types, and system composition in a real-world CPS. By modeling the system's behavior, formalizing interaction protocols, and enabling dynamic reconfiguration, the framework provides a robust foundation for adapting to complex and evolving requirements.

V. DEVICE CAPABILITY ONTOLOGY

Traditional transition systems and model-checking approaches effectively verify control software correctness, focusing on predefined state transitions. However, they often lack the flexibility for real-time adaptation in dynamic and evolving CPSs, where new devices and capabilities may be introduced at runtime.

A. Transition Systems and Capability Ontology

While traditional transition systems and model-checking approaches are valuable for verifying control software correctness, they have inherent limitations when applied to dynamic and evolving CPSs. Transition systems focus on predefined state transitions and require exhaustive state-space exploration to ensure correctness. This is effective in static environments but becomes challenging in CPS, where new devices and capabilities may be introduced or modified at runtime, leading to frequent changes in system behavior.

The proposed ontology-based approach extends the strengths of transition systems by incorporating their structural elements, such as states, transitions, inputs, and outputs, while adding the flexibility and expressiveness needed for real-time reconfiguration. The ontology framework allows the system to capture static and dynamic aspects of device capabilities and interactions, supporting more adaptable and scalable representations. The key reasons for favoring ontologies over model-driven approaches are as follows.

1) *Dynamic Representation:* Ontologies enable the representation of both static and dynamic aspects of a system. While the structure of a transition system is incorporated into the ontology to model states and transitions, the ontology extends this model by allowing for the dynamic addition and modification of device capabilities and relationships without the need for manual reconfiguration. This is crucial in CPS environments where the system must adapt quickly to evolving operational conditions.
2) *Semantic Reasoning and Inference:* The ontology supports semantic reasoning, enabling the system to infer new knowledge based on defined relationships and properties. This capability goes beyond the limitations

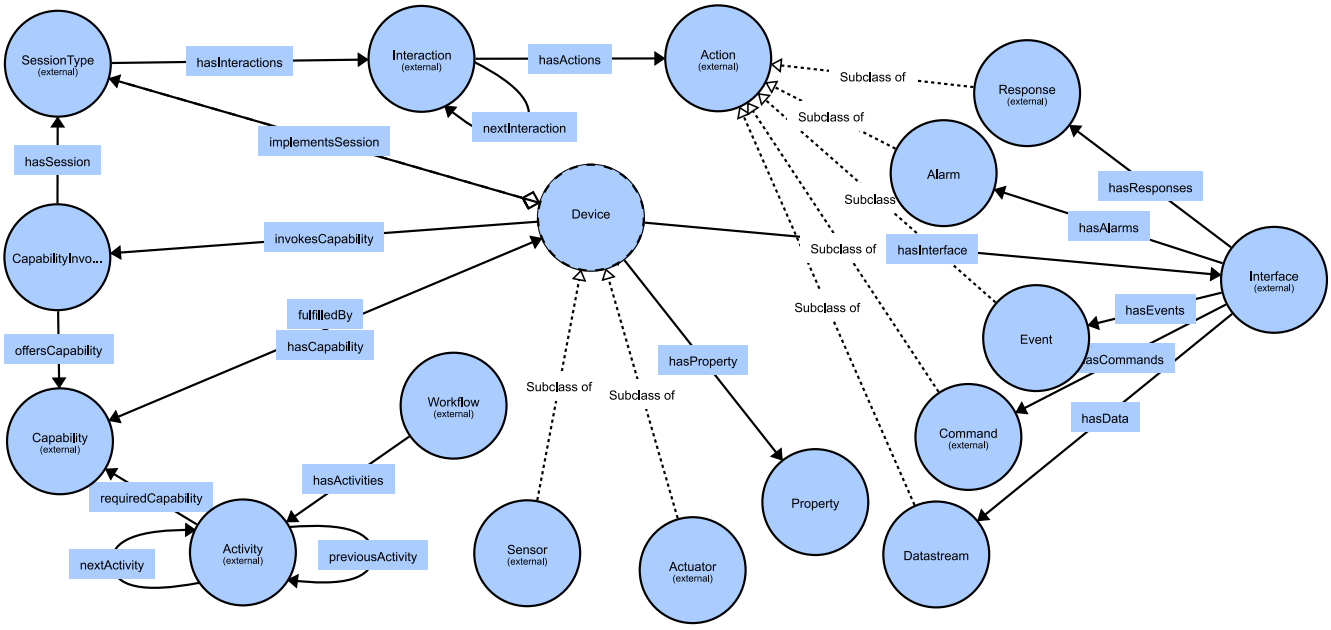


Fig. 3. Capability ontology.

of transition systems. For example, if a new device with specific capabilities is added, the ontology-based approach can automatically infer how this device fits into existing workflows, thus facilitating efficient reconfiguration.

3) *Heterogeneity Handling:* CPS environments often consist of heterogeneous devices with different capabilities and interfaces. The ontology provides a standardized way to represent various device types, functionalities, and interactions while incorporating the structural elements of the transition system. This ensures interoperability and scalability across a diverse set of devices.
4) *Reduced Computational Overhead:* Model-checking requires exhaustive exploration of all possible states and transitions, which can be computationally intensive. The ontology approach incorporates transition system structures and uses inference rules and semantic querying to focus on relevant updates and adaptations rather than exploring the entire state space. This significantly reduces computational overhead in large-scale environments.

The ontology proposed in this article, shown in Fig. 3, is designed to represent capability knowledge comprehensively. It extends concepts from the iotschema.org ontology [27] to address identified gaps, particularly by capturing static device capabilities and dynamic aspects, such as session types and workflows, essential for real-time reconfiguration in CPS.

B. Classes

The ontology introduces several classes to represent entities within CPS. The `Device` class, with subclasses `Sensor` and `Actuator`, models the physical components of an IoT system, capturing unique characteristics and capabilities. This aligns with the formal definition of a plant, where each device represents a system with its own state space, input set, transition relation, and output set.

TABLE IV
CLASSES IN THE CAPABILITY ONTOLOGY

Class	Description
Action	Actions a device can perform :- Command trigger, Response/Event/Alarm/Data handling, Event/Alarm/Data generation, State Transition
Activity	A sequence of actions in a workflow
Device	A device with certain capabilities
Capability	A functionality a device can offer
Interface	An interface for device interaction
SessionType	A type of session a device can implement
Workflow	A workflow for dynamic reconfiguration

These classes provide a detailed and flexible representation of device capabilities, supporting dynamic reconfiguration and adaptation to real-time conditions.

The ontology is designed to evolve, supporting long-term monitoring and adaptation by updating its representation of device capabilities and system requirements. This ensures the control system remains accurate and responsive to new devices, technologies, and operational conditions, with inference rules enabling the automatic discovery of new capabilities and workflows.

C. Properties

Object properties define relationships between instances of these classes. For instance, the `hasCapability` property links a `Device` to a `Capability`, and the `hasActivities` property links a `Workflow` to an `Activity`. These properties capture the relationships between entities, offering a rich and flexible representation of capability knowledge.

These properties enable the control system to dynamically reconfigure itself by understanding and leveraging the relationships between devices, capabilities, and workflows.

TABLE V
OBJECT PROPERTIES IN THE CAPABILITY ONTOLOGY

Object Property	Description
fulfilledBy	Relates a capability to the device or session type that fulfills it
hasActivities	Relates a workflow to its activities
hasCapability	Relates a device to its capabilities
hasInteractions	Relates a session type to its interactions
hasInterface	Relates a device to its interface for interaction
implementsSession	Relates a device to the session types it can implement
requiredCapability	Relates an activity to the capabilities required

D. Rules

Inference rules, particularly those defined in semantic Web rule language (SWRL) [28], enhance the expressivity and utility of the capability ontology. These rules automate the reasoning process, enabling the control software to make informed decisions about reconfiguration based on current conditions and requirements.

An Example Rule: "If a device invokes a capability offered by a certain workflow, then the device has that capability," is represented in SWRL as:

```
Device(?d) ^ CapabilityInvocation(?ci)
^ Capability(?c) ^ invokesCapability(?d,
?ci) ^ offersCapability(?ci, ?c) ->
hasCapability(?d, ?c)
```

Such rules are crucial for enabling real-time reconfiguration by allowing the control software to automatically infer new capabilities and workflows, ensuring quick and accurate adaptation to environmental changes.

E. Comparison of Capability Ontology With Other Ontologies

To demonstrate the advantages of the proposed capability ontology, we provide a comparative analysis with existing ontologies, focusing on interface representation, behavior modeling, workflow inclusion, inferencing capabilities, and support for dynamic reconfiguration. The comparison in Table VIII illustrates existing ontologies' limitations and highlights the capability ontology's unique features that address these gaps.

Alsafi and Vyatkin [19] proposed an ontology-based reconfiguration for mechatronic systems with basic support for reconfiguration. However, it lacked comprehensive workflow integration and real-time inferencing capabilities, limiting its flexibility in dynamic environments. Pentga and Austin [18] introduced a CPS decision-support framework that included behavior modeling and inferencing but did not consider interaction modeling or dynamic workflows, which are essential for comprehensive reconfiguration. However, it lacked dynamic reconfiguration capabilities and did not integrate workflows for system adaptation.

Wan et al. [16] developed an ontology-based resource reconfiguration approach for manufacturing CPSs, providing structured modeling but limited real-time adaptability and no workflow integration, hindering frequent dynamic reconfiguration. Nguyen-Anh and Le-Trung [15] combined ontology and Bayesian networks for IoT reconfiguration. It provided interaction modeling and partial support for reconfiguration but lacked a comprehensive workflow approach and had complexity in setting up Bayesian networks.

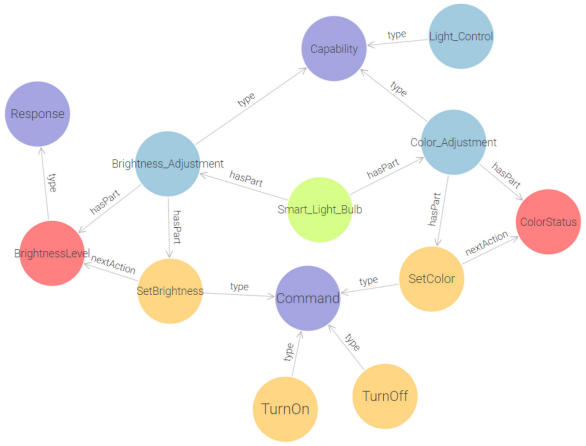


Fig. 4. Persistence of device knowledge in a graph database.

The MaRCO ontology [17] standardized manufacturing resource capabilities with detailed behavior modeling but lacked support for workflow-based dynamic reconfiguration. The C4I ontology [13] utilized Semantic Web technologies, providing some inferencing and behavior modeling, but had limited dynamic reconfiguration support and did not fully integrate workflow management. Sari et al. [14] focused on the runtime binding of IoT applications but did not extend to system-level reconfiguration. It had limited inferencing capabilities and did not support complex workflows.

Thuluva et al. [12] extended Node-RED for IoT with semantic modeling, including partial interaction and inferencing capabilities, but lacked full reconfiguration and workflow integration support. Fadhlillah et al. [7] proposed a variability management approach for CPPSs with partial support for dynamic reconfiguration. However, it did not fully address workflow integration and had limited inferencing capabilities.

The Capability Ontology addresses these limitations by supporting comprehensive modeling of interfaces, behaviors, interactions, workflows, and real-time inferencing. It offers persistence and enables dynamic reconfiguration, providing a complete solution for adaptive control software in complex environments. This analysis shows that the proposed capability ontology offers comprehensive support for dynamic reconfiguration, addressing the limitations of existing ontologies. By supporting interfaces, behaviors, interactions, workflows, and inferencing, our ontology provides a robust framework for managing the complexities of CPS environments.

F. Persistence in Graph Database

Fig. 4 illustrates the persistence of individuals in our ontology in a graph database, offering advantages for representing and querying capability knowledge. Graph databases like GraphDB and Stardog are particularly well-suited for this task due to their support for graph-based data models and semantic Web technologies.

TABLE VI
COMPARISON OF ONTOLOGIES FOR DYNAMIC RECONFIGURATION IN CONTROL SOFTWARE

Ontology	Year	Interface	Behavior	Interaction	Workflow	Inferencing	Persistence	Reconfiguration
Fadhlillah et al. [7]	2022	Yes	Yes	Yes	No	Partial	Yes	Partial
C4I [13]	2020	Yes	Yes	Partial	No	Partial	Yes	Partial
Sari et al. [14]	2020	Partial	No	Yes	No	No	Yes	Partial
Thuluva et al. [12]	2020	Partial	No	Yes	No	Partial	Yes	No
Nguyen-Anh et al. [15]	2019	Yes	Partial	Yes	No	Yes	Yes	Partial
MaRCO [17]	2019	Yes	Yes	Partial	No	Partial	Yes	No
Wan et al. [16]	2018	Yes	Yes	Partial	No	No	Yes	No
Pentga et al. [18]	2016	Yes	Yes	No	No	Yes	Yes	Partial
Capability Ontology	2023	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Alsafi et al. [19]	2010	Yes	Partial	No	No	Partial	Yes	Partial

Graph databases represent entities as nodes and relationships as edges, creating a graph-based structure that closely matches our ontology. This representation facilitates efficient querying and mapping of the ontology to the database.

By persisting the ontology in a graph database, we ensure that the control software can quickly and efficiently access and update capability knowledge, enabling real-time dynamic reconfiguration in CPS environments. This approach also supports scalability, allowing the ontology to handle large volumes of data and complex queries without performance degradation.

In conclusion, the proposed capability ontology provides a comprehensive framework for representing and utilizing device capabilities in CPS. By incorporating classes, properties, inference rules, and persistence mechanisms, the ontology enables dynamic reconfiguration, ensuring the system can adapt to changing conditions and maintain high performance and reliability.

G. Real-Time Updates and Scalability of the Capability Ontology

The capability ontology supports real-time updates by triggering dynamic reconfiguration in response to the continuous evolution of device capabilities in a CPS. The system uses event-based triggers to detect changes in the network, such as adding new devices, removing existing ones, or modifying their capabilities. Upon detecting a new device, the system performs a capability profile query (e.g., fetching supported actions, interfaces, and state transitions), which is then used to update the ontology accordingly. If changes in device capabilities are detected, only the relevant entries in the ontology are modified, ensuring consistency without a complete ontology rebuild.

The system performs incremental updates and differential querying [29] techniques to optimize computational performance. Incremental updates allow only the modified portions of the ontology to be updated, thereby avoiding a complete reprocessing of the knowledge base. Differential querying compares the current ontology state with the new updates and executes SPARQL update queries to modify specific instances and relationships in the ontology. This selective update approach significantly reduces the computational cost, ensuring the system remains responsive under varying loads. These methods support real-time adaptation while maintaining data integrity.

Scalability is also addressed through architectural choices. The ontology is partitioned logically and functionally to distribute the processing load efficiently. For example, ontology segments are categorized based on device types (e.g., sensors, actuators) or functional groups (e.g., control nodes, monitoring nodes). Partitioning facilitates distributed reasoning across multiple processing nodes, enabling parallel updates and reducing the latency of querying large knowledge graphs. Furthermore, batch processing is used to handle scenarios involving simultaneous updates, where related modifications are grouped to minimize the number of individual operations.

Finally, the system also employs caching mechanisms to improve querying efficiency for large-scale CPS environments, such as smart factories with thousands of devices. Frequently accessed data, such as commonly queried device capabilities and control workflows, are cached in memory, reducing the need for repeated complex queries against the graph database. The ontology's persistence layer uses graph databases like GraphDB, which supports RDF-based data models and SPARQL for efficient querying and reasoning over large datasets. To further enhance persistence performance, the graph database employs indexing strategies that optimize query execution times, especially for complex SPARQL queries that involve multiple relationships and properties.

The system's scalability is validated through performance metrics, including latency, update response times, and computational overhead, which are discussed in detail in the subsequent evaluation in Section X. For instance, the average computational overhead for adding a new device ranges from 50 to 100 ms, depending on the complexity of the device's capability profile. In comparison, updates to existing capabilities typically require between 30 and 80 ms.

VI. COMPOSITION ARCHITECTURE

The sensor actuator control element (SACE) [30], [31] architecture, depicted in Fig. 5, is a model-driven framework designed to generate control and monitoring software for large real-time systems, such as radio telescopes. SACE supports scalability and adaptability, enabling multiple devices and subsystems to seamlessly integrate into a cohesive control system capable of dynamic reconfiguring in response to changing conditions.

The SACE architecture integrates control nodes in a recursive hierarchical manner, mirroring the hierarchical structure

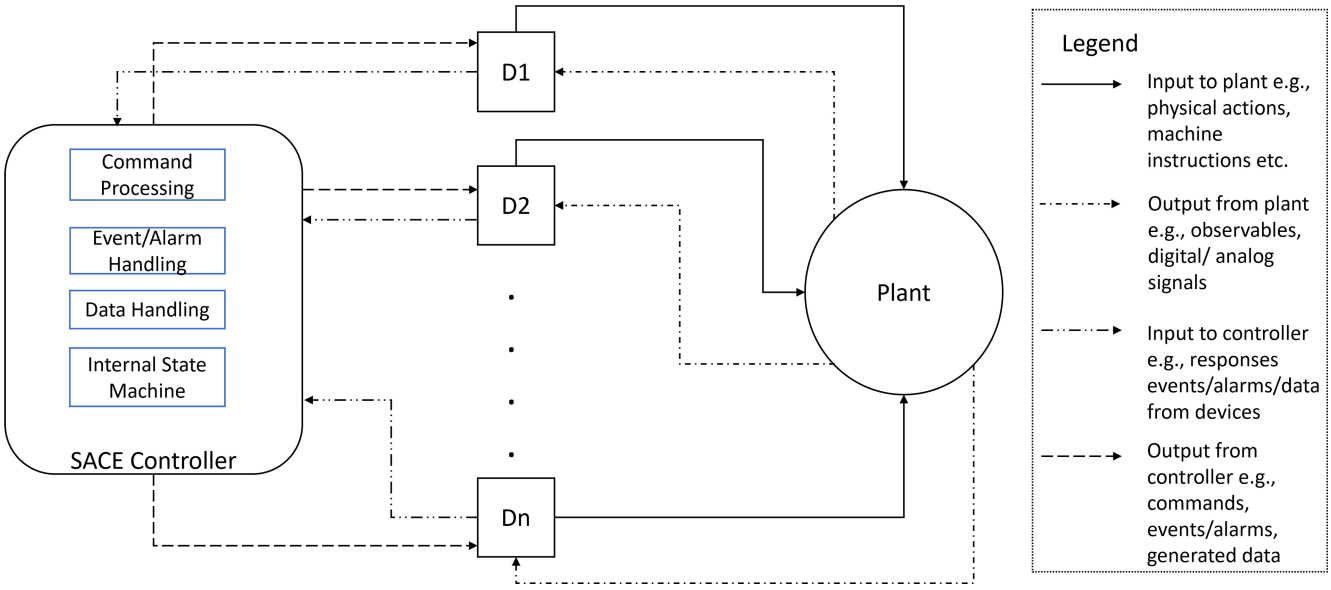


Fig. 5. SACE architecture.

of the proposed capability ontology. Each control node coordinates its subsystems, analogous to how the `Device` class in the ontology coordinates its `Capabilities`. Control software for each node is generated from specifications of control logic, data acquisition and processing, and event handling, paralleling the ontology's definition of class properties and rules.

The SACE architecture's recursive hierarchical design is key to its flexibility, allowing the seamless integration of new devices and subsystems. This design ensures the control system remains scalable and adaptable, managing complex configurations and dynamically reconfiguring itself as needed. The architecture's versatility suits various CPS domains, such as industrial automation, smart grids, and autonomous vehicles, where systems must adapt to diverse and evolving operational conditions.

The SACE Node, the basic building block of the framework, provides three functional interfaces to higher and lower levels: 1) command; 2) data; and 3) event interfaces. These interfaces facilitate the hierarchical composition of SACE nodes into a system, reflecting the recursive structure of the ontology.

These interfaces enable dynamic reconfiguration by allowing the control software to coordinate the actions of various subsystems and adapt to environmental or operational changes. For example, in industrial automation, the command interface executes control commands for machinery, the data interface processes sensor data, and the event interface manages fault detection and response. This capability is equally applicable in smart grids, where SACE dynamically balances load distribution based on real-time data, or in autonomous vehicles, where it coordinates navigation and obstacle avoidance.

Each SACE Node's internal architecture comprises three main paths: 1) command; 2) data; and 3) event. The command path validates and executes commands, orchestrating actions through a finite-state machine (FSM). The data path handles acquisition, processing, and streaming, ensuring real-time data availability for decision-making. The event path manages event acquisition and distribution, enabling prompt system responses to changes or anomalies. These paths work together to maintain high-system performance and reliability, even during real-time reconfiguration.

In control software reconfiguration, the SACE architecture provides a flexible framework for adapting to changing control requirements. This aligns with the ontology's focus on dynamic reconfiguration, reducing development time and effort while simplifying the evolution of control software as it changes.

Integrating the SACE architecture with the capability ontology enhances the system's dynamic reconfiguration management. By leveraging detailed capability knowledge from the ontology, SACE effectively coordinates subsystem actions, ensuring quick and efficient adaptation to new requirements or environmental changes. For instance, SACE can dynamically reconfigure production workflows based on real-time machinery data in industrial automation. In smart grids, it adjusts power distribution in response to fluctuating demand, and in autonomous vehicles, it adapts navigation strategies based on sensor data and environmental changes.

The proposed ontology leverages various aspects of the SACE architecture to define the capability ontology, providing a structured and hierarchical information flow that facilitates efficient decision-making and control throughout the plant or system. The architecture's central intelligence, in the form of a supervisory controller, receives information from lower levels, analyzes it, and provides control signals to drive overall plant operation. This hierarchical structure facilitates effective coordination among multiple devices or subsystems, improving performance, fault tolerance, and scalability.

A key advantage of the SACE architecture is its support for scalability and adaptability. Organizing control nodes hierarchically allows the architecture to easily accommodate new devices or subsystems without extensive changes to existing

control software. This flexibility is crucial for managing the complexity of modern CPS environments, where devices and subsystems are frequently added, removed, or reconfigured. The architecture's adaptability ensures it can be applied across various CPS domains, from industrial automation to smart grids and autonomous vehicles, making it a versatile tool for dynamic reconfiguration.

The next section discusses an algorithm for composition, further enhancing the SACE architecture's flexibility and adaptability in control software reconfiguration.

VII. Composing Higher Order Machines

One of the key strengths of our ontology is its ability to facilitate the composition of higher order machines based on workflow and capability knowledge. This is achieved through a scalable algorithm that dynamically identifies and composes devices capable of fulfilling specific workflow requirements, ensuring the control system can adapt to real-time changes in various CPS domains, such as industrial automation, smart grids, and autonomous vehicles. To this end, we developed an algorithm that utilizes SPARQL queries to identify devices offering specific capabilities and session types to invoke those capabilities. The algorithm then constructs a controller based on the sense-analyze-compose-execute (SACE) architecture, integrating the identified devices to fulfill the desired workflow.

A. SPARQL Queries for Device Capabilities

The algorithm begins with SPARQL queries to retrieve devices that offer a particular capability. This step aligns with the `Device` class in our ontology, representing a device with specific capabilities. By querying the graph database, we can match devices with the desired capability and obtain a list of devices along with their associated properties and relationships.

This querying process is optimized for scalability, enabling the algorithm to handle large data volumes efficiently. SPARQL queries ensure the system can quickly identify the most suitable devices for a given task, even in complex and dynamic CPS environments. For example, in an industrial automation setting, the algorithm can swiftly select machinery based on real-time sensor data. At the same time, in smart grids, it can dynamically allocate resources based on current power demands.

B. Session Types for Capability Invocation

Once the devices offering the desired capability are identified, the algorithm employs session types to invoke the capability. This step corresponds to the `SessionType` class in our ontology, representing a type of session that can be implemented by a device in the control software. Session types formalize communication protocols, specifying the sequence of messages exchanged between the controller and the device to invoke the capability effectively.

By formalizing session types, the algorithm ensures that the control system can coordinate complex interactions between devices in real time. This is crucial for maintaining system reliability and performance during dynamic reconfiguration. For instance, the algorithm coordinates real-time sensor inputs and actuator responses in autonomous vehicles to ensure safe navigation and obstacle avoidance.

Session types provide a theoretically sound mechanism for ensuring error-free communication protocols in the control system by specifying the expected sequence of interactions [32]. They enable type matching between the controller and device interfaces, ensuring that only compatible communication protocols are executed, thereby offering a correct-by-design approach [32]. This formalism guarantees that the system adheres to predefined interaction patterns, preventing communication errors during capability invocation [32].

However, a limitation exists when the capability required for reconfiguration is absent in the knowledge base. In such cases, the system may be unable to match any session type, resulting in potential stalling or delays in reconfiguration. The architecture includes fallback strategies to address this, such as using default or alternative workflows, when an exact capability match is unavailable. Error-handling mechanisms are in place to log such occurrences and notify operators, allowing for manual intervention.

Empirically, across the use cases discussed in later sections (e.g., Smart Meeting Room, Robotic Arm Programming, and Vehicle Gate Automation), no errors related to session type mismatches were observed, demonstrating the robustness of this approach in practical scenarios.

C. Building the Controller

After identifying the devices and session types, the algorithm constructs the controller. The controller is built based on the SACE architecture, using a reasoning mechanism to find the best device offering a matching capability for each activity in the *Workflow*. This reasoning process optimizes the controller's composition, balancing factors, such as latency, accuracy, and resource availability, to ensure the control system can respond effectively to real-time changes. For example, the algorithm optimizes load distribution in a smart grid scenario by selecting the most efficient combination of power sources and storage systems. Such compositional reasoning [33] allows us to derive multiple controllers for the same problem, which can then be evaluated based on cost, procurement time, maintenance efforts, etc.

D. Integrating Semantic Rules of the Transition System

Incorporating the semantic rules of the transition system into our algorithm is crucial for ensuring that the generated controllers respect the system's operational semantics. Each transition in a traditional transition system can be mapped to a session type in our ontology, ensuring that the sequence of actions taken by the control system adheres to the expected behavior as defined by the transition semantics.

The transition system's states correspond to specific configurations or states of the devices in the CPS, while the transitions represent the invocation of capabilities or changes in the workflow. By ensuring that each state transition in the workflow corresponds to a valid session type and device

capability, the algorithm maintains the integrity of the system's operational semantics.

Formally, the process is described as follows.

1) *State Representation:* Each state in the transition system is represented by a tuple of device configurations and operational conditions.
2) *Transition Semantics:* Transitions between states are governed by session types that define the sequence of actions (capabilities) required to move from one state to another.
3) *Soundness Proof:* The algorithm guarantees that for any valid sequence of transitions (as defined by the transition system), a corresponding sequence of session types and capabilities exists in the generated controller, ensuring the system behavior adheres to the specified semantics.

E. Integrating the Devices

The final step of the algorithm involves integrating the various devices offering the capability to fulfill the workflow. This is achieved by establishing communication links between the controller and the devices and coordinating their actions to achieve the desired workflow. This step aligns with the `Interface` class in our ontology, which represents the interface of a device that allows interaction with the control software.

The integration process is designed to be flexible and adaptive, ensuring that the control system can incorporate new devices and reconfigure existing ones with minimal disruption. This flexibility is crucial for maintaining system performance and reliability in dynamic CPS environments. For instance, in industrial automation, the integration allows for the seamless addition of new machinery or sensors without interrupting ongoing production processes.

The algorithm for composing controllers based on device capabilities and session types is shown in Algorithm 1.

Algorithm 1 Algorithm for Composing Controllers

```
function ComposeMachines(machines, capabilities, session types)
    seq_list ← list of machines with sequential capabilities
    par_list ← list of machines with parallel capabilities
    composed_seq ← empty list
    for all m ∈ seq_list do
        m_session ← session type of machine m
        m_capability ← capability of machine m
        if previous machine in composed_seq has output that satisfies m_capability then
            append m to composed_seq
        else
            session_input ← input from the previous machine's output
            session_output ← run m_session with session_input
            if session_output satisfies m_capability then
                append m to composed_seq
            else
                raise SessionTypeError
            end if
        end if
    end for
    for all m ∈ par_list do
        m_session ← session type of machine m
        m_capability ← capability of machine m
        session_input ← input from the previous machine's output
        session_output ← run m_session with session_input
        if session_output satisfies m_capability then
            append m to composed_seq
        else
            raise SessionTypeError
        end if
    end for
    return composed_seq
end function
```

F. Mapping From Transition Systems to Session Types

Let S be a plant defined as a transition system in Definition 1. Let T be a set of session types in Definition 5, P be a set of participant names, $p \in P$, and M be a set of message types (commands).

Mapping Function: Define a mapping function $\phi{:}U_S \to T$ that maps each action in the transition system to a session type in the control software.

Lemma 1: The mapping ϕ is sound *iff* $\forall(x, u, x') \in S$, the session type $\phi(u)$ results in a transition from a state corresponding to x to a state corresponding to x'.

Proof: We prove the soundness of the mapping function ϕ by induction on the structure of the transition system.

Base Case: Consider the initial state $x_0 \in X_S^0$. Assume that x_0 is mapped to the initial configuration *Init* of the session type, i.e., $\phi(Init) = x_0$. This trivially satisfies the soundness criterion, as there are no prior transitions.

Inductive Step: Assume that for any state x reachable from x_0 through a sequence of actions $u_1, u_2, \ldots, u_n$, the corresponding session type sequence $\phi(u_1), \phi(u_2), \ldots, \phi(u_n)$ maps x to the correct final state x'.

Now, consider an action u_{n+1} leading from state x to x_{n+1}. By the session type construction, the sequence $\phi(u_{n+1})$ ensures that the state transition $x \xrightarrow{u_{n+1}} x_{n+1}$ is preserved.

By induction, the session type sequence preserves the transition semantics of the original transition system. Therefore, the mapping function ϕ is sound, meaning that the composed controller generated by the algorithm respects the operational semantics of the system as defined by the transition system.

Formally $\forall(x, u, x') \in S$, the session type $\phi(u)$ leads the system from a state corresponding to x to a state corresponding to x'. Hence, ϕ is a sound mapping from transition systems to session types. ■

G. Implementation of the Algorithm With Semantic Rules and Formal Description

The implementation of the algorithm for dynamic reconfiguration in the control system leverages semantic rules and

a formal mapping between the transition system and session types. The algorithm ensures that each state transition in the system corresponds to an appropriate invocation of a session type, maintaining the integrity of the system's operational semantics. Below, we describe the key elements of the implementation.

1) Semantic Rules Using SWRL: The algorithm utilizes the SWRL to encode the semantic rules that govern state transitions. These rules specify how different elements of the transition system map to session types in the ontology. For instance, if a transition requires a device to perform a certain action (capability), an SWRL rule verifies that the device's capability matches the required action before invoking the session type.

An example SWRL rule to map a transition to a session type is:

```
Transition(?t) ^ Capability(?c) ^ requires
Action(?t, ?a) ^ providesCapability(?c, ?a)
```

This rule ensures that for every transition requiring an action a, there is a corresponding capability c that can act, thereby mapping the transition to the correct session type.

2) Formal Mapping Description: To formally describe the mapping, we define a function ϕ that connects the elements of the transition system to session types in the control software

$$\phi : U_S \rightarrow T \tag{10}$$

where U_S is the set of actions in the transition system and T is the set of session types.

The mapping ensures that for every transition $(x, u, x') \in S$, there exists a session type $\phi(u)$ that corresponds to moving from state x to x'. The algorithm ensures that each mapped session type satisfies the conditions specified by the transition, preserving the intended behavior of the system.

3) Mapping Logic: We prove the soundness of the mapping function ϕ using induction.

1) *Base Case:* The initial state x_0 is mapped to the initial configuration of the session type. The SWRL rules ensure that this initial mapping is valid.
2) *Inductive Step:* The mapping is verified using SWRL rules, confirming that all conditions for the transition are met as per Lemma 1.

By following this process all generated controllers adhere to the system's operational semantics as defined by the original transition system.

4) Algorithm Workflow: The algorithm for composing controllers based on semantic rules and the formal mapping proceeds as follows.

1) *Query the Ontology:* Use SPARQL queries to identify devices that offer capabilities matching the actions required by the transitions.
2) *Apply SWRL Rules:* Evaluate the SWRL rules to validate that the identified capabilities match the required session types.
3) *Map Transitions to Session Types:* Use the mapping function ϕ to associate each transition with a corresponding session type.
4) *Construct the Controller:* Assemble the session types into a sequence that forms a complete controller, ensuring that the system's state transitions follow the original operational semantics.

This implementation ensures that the algorithm maintains the integrity of the transition system while enabling dynamic reconfiguration in complex CPS environments. The approach offers a robust mechanism for adapting to changing operational conditions by integrating semantic rules and providing a formal soundness guarantee.

This algorithm is critical in enabling the dynamic reconfiguration of control systems. By leveraging the detailed capability knowledge encoded in the ontology and the formalism of session types, the algorithm ensures that the control system can quickly and efficiently reconfigure itself to meet new operational demands.

H. Fallback Strategies and Error-Handling Mechanisms

The algorithm incorporates fallback strategies and error-handling mechanisms to handle situations where the required capability is absent in the knowledge base or the session type does not match. This ensures robust dynamic reconfiguration, even when encountering unexpected conditions. The pseudocode below outlines these strategies.

1) The function `InvokeCapabilityWithFallback` handles the capability invocation process while providing fallback strategies for error scenarios.
2) If the required capability does not exist, it first attempts to use a default workflow for the device. The operator is notified for manual intervention if a default workflow is unavailable.
3) The algorithm checks for an alternative workflow if the capability exists, but the session output does not satisfy the capability requirements. The operator is again notified of manual intervention if no alternative is available.
4) All errors and warnings are logged for traceability and debugging purposes.

VIII. IDE Platform for Automated Control Design and Dynamic Reconfiguration

This section introduces an IDE platform to automate control design and enable dynamic reconfiguration across various applications. The platform integrates our device-capability ontology, the SACE architecture, and an algorithm for composing higher order machines to streamline control system development.

A. Platform Overview

The IDE platform is a comprehensive tool for control system engineers, offering features that facilitate control design, analysis, and reconfiguration. By utilizing capability knowledge, the platform automates several aspects of control system development, including device selection, control strategy formulation, and system reconfiguration [34]. Fig. 6 illustrates the platform's modules, which enable knowledge-driven controller synthesis and reconfiguration.

The platform emphasizes scalability and adaptability, ensuring it can manage complex control systems and respond to

Algorithm 2 Fallback Strategies and Error Handling for Capability Invocation

```
function INVOKECAPABILITYWITHFALLBACK(device, capability, sessionType)
    if not CAPABILITYEXISTS(device, capability) then ▷ Check if the capability exists in the knowledge base
        LOGERROR("Capability not found: " + capability)
        if HASDEFAULTWORKFLOW(device) then ▷ Check for default workflows as fallback
            fallbackSession ← GETDEFAULTWORKFLOW(device)
            LOGWARNING("Using default workflow for device: " + device)
            INVOKEFALLBACKSESSION(device, fallbackSession)
        else ▷ No fallback available, notify operator
            NOTIFYOPERATOR("Manual intervention required for device:" + device)
            return False
        end if
    else ▷ Capability exists, proceed with invocation
        sessionOutput ← INVOKESESSIONTYPE(device, sessionType)
        if not OUTPUTSATISFIESCAPABILITY(session Output, capability) then ▷ Check if output satisfies the capability requirements
            LOGERROR("Session output does not satisfy capability requirements")
            if HASALTERNATIVEWORKFLOW(device) then ▷ Try an alternative workflow if available
                alternativeSession ← GETALTERNATIVEWORKFLOW(device)
                LOGWARNING("Using alternative workflow for device: " + device)
                INVOKEFALLBACKSESSION(device, alternativeSession)
            else ▷ No suitable alternative, notify operator
                NOTIFYOPERATOR("Manual intervention required for device: " + device)
                return False
            end if
        else ▷ Successful invocation
            return True
        end if
    end if
end function
```

changing requirements in real time. By integrating the device-capability ontology with the SACE architecture, the platform provides a robust foundation for dynamic reconfiguration, allowing control systems to adapt to new operational demands quickly.

Grounded in a KDA [23], the IDE platform utilizes a KDA stack. Various modules are implemented using the technology stack shown on the left in Fig. 6, with all components integrated within the Eclipse framework [35] to support scalability and modularity.

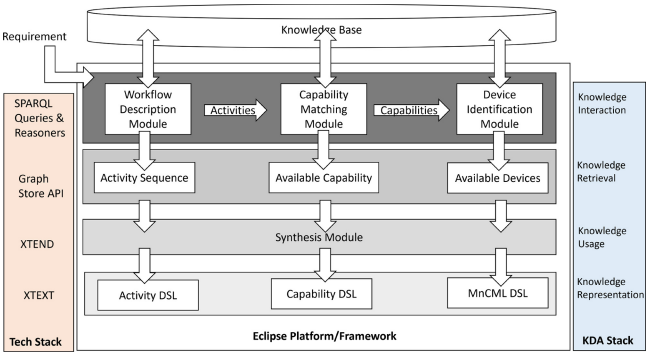


Fig. 6. Capability-driven ide for control software development.

B. Automated Control Design and Reconfiguration

The platform's automated control design capabilities are powered by the device-capability ontology and the algorithm for composing higher order machines. Users can define control objectives using a domain-specific language (DSL) [36] and specify the desired workflow within the IDE [10]. The platform then identifies devices offering the required capabilities and session types, automatically composing the controllers based on the SACE architecture.

K-IDE uses multiple DSLs to facilitate the automation of control system development and dynamic reconfiguration. The following steps illustrate the K-IDE workflow using DSLs, supported by figures demonstrating the transformation from high-level use cases to executable code.

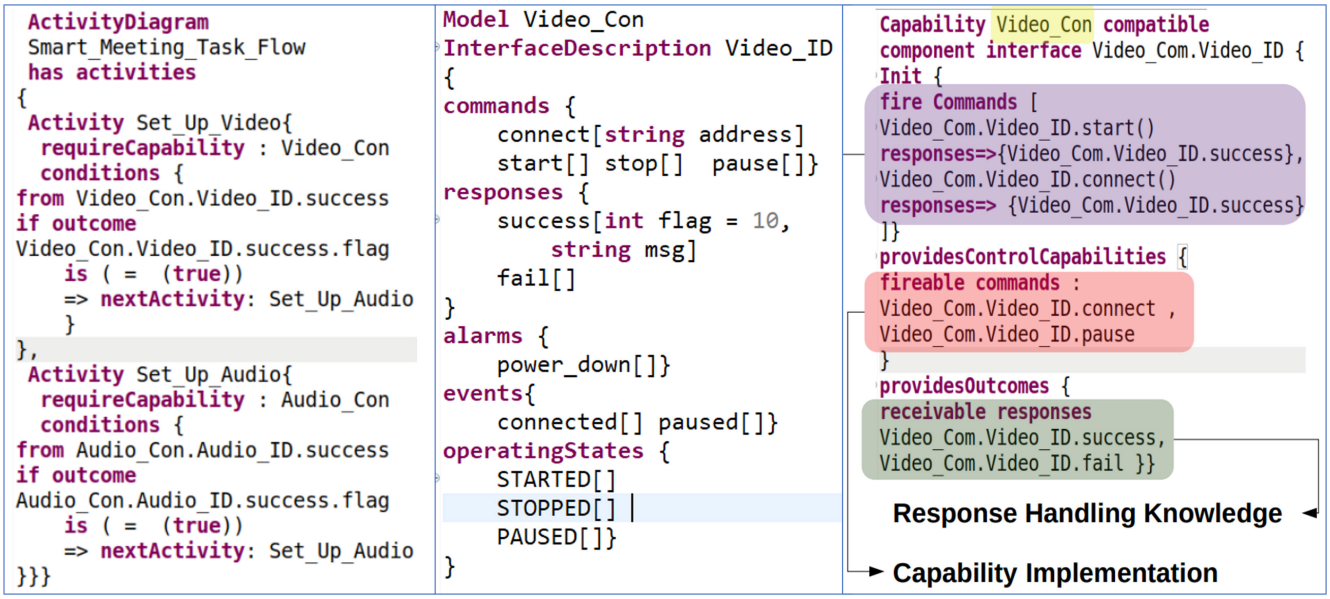
1) Step 1—Activity Workflow Creation Using Activity DSL: The *Activity DSL* is then employed to map high-level control objectives to executable workflows, decomposing activities into tasks. Activities are recursively broken down until they can be linked to specific capabilities in the system. Fig. 7(a), shows an example of describing control recipe knowledge for a smart meeting room using Activity DSL.

2) Step 2—Create Device Knowledge Using M&CML: The first step involves capturing detailed descriptions of device capabilities using the monitoring and control modeling language (M&CML) [36]. This DSL helps define the interface, behavior, and states of each device, modeling resource capabilities. Fig. 7(b), shows an example of describing device interface knowledge using M&CML dsl.

3) Step 3—Capability Description Using Capability DSL: Next, the *Capability DSL* is used to catalogue system component capabilities, defining their interactions, behavior, actions, and outcomes. This step standardizes the representation of device capabilities, as shown in Fig. 7(c).

4) Step 4—Knowledge Persistence in the Knowledge Graph: The compiled knowledge, including device descriptions, capabilities, and workflows, is persisted in a *Knowledge Graph* using GraphDB. This ensures a structured and retrievable format for future use.

5) Step 5—Code Generation Using KDL DSL: The *knowledge description language (KDL)* is used to transform the



(a) (b) (c)

Fig. 7. Application of K-IDE DSLs in domain knowledge representation. (a) Description of a smart meeting room control recipe described using activity DSL. (b) Description of a device interface for a video conference system in a smart meeting room scenario using M&CML DSL. (c) Description of the video conference device capability using capability DSL.

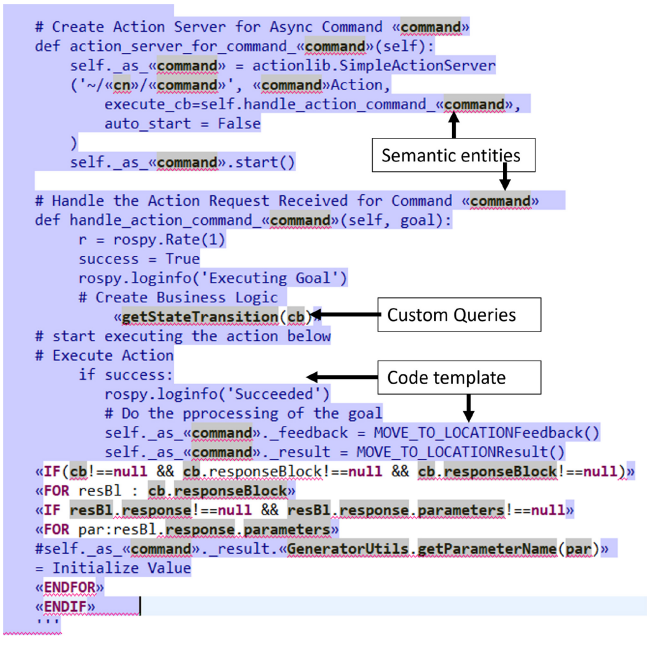


Fig. 8. Generating executable code using KDL DSL.

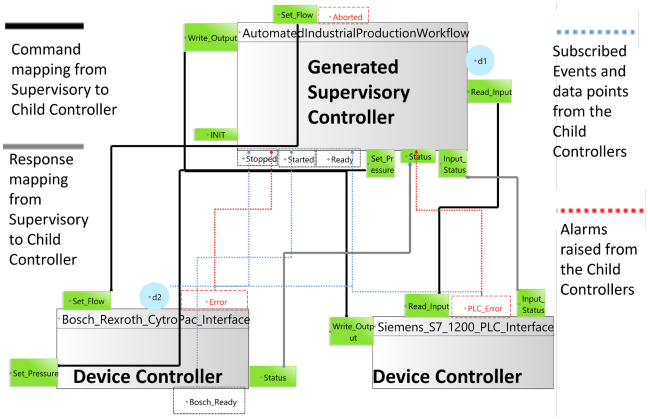


Fig. 9. Example of visual representation in K-IDE: interconnections between supervisory and device controllers.

stored knowledge into executable code. KDL maps semantic entities from the Knowledge Graph to code templates, generating code tailored to specific project requirements.

6) Step 6—Dynamic Reconfiguration and Testing: The K-IDE continuously monitors the system state and reconfigures control strategies based on triggers defined in the MnCML. The generated code is tested and verified under different scenarios to ensure system robustness.

7) Step 7—Visualization of Generated Design: K-IDE finally visualizes the generated control design using visualization techniques, such as model-to-model transformations [37].

The workflow demonstrates how K-IDE integrates Activity DSL, M&CML, Capability DSL, and KDL to automate the development and reconfiguration processes, facilitating efficient and reliable control system development.

This approach offers significant advantages.

1) *Knowledge Reuse:* The platform ensures that generated code is grounded in the most current and accurate information by reusing domain knowledge.
2) *Flexibility:* Using a custom DSL allows for creating mappings tailored to the domain's specific needs. As

the domain evolves, mappings can be easily updated or expanded.

The code generation approach has been successfully applied in various use cases, including digital twins [24], [38] and automated control software development scenarios [10], [23].

C. Security Implications and Mitigation Strategies for IDE

Dynamic reconfiguration in CPSs, especially those involving sensitive or critical infrastructure, presents security challenges. Ensuring reconfigurations do not introduce vulnerabilities is crucial for system integrity and safety. The IDE integrates multiple layers of security to mitigate risks, as described below.

1) Secure Knowledge Persistence in Graph Datastore: The graph-based datastore (e.g., GraphDB) enforces strict authentication and authorization for accessing and modifying stored knowledge, preventing unauthorized changes.

1) *Role-Based Access Control (RBAC):* Allows fine-grained access, limiting permissions based on user roles (e.g., read-only or administrative).
2) *Audit Logs for Knowledge Changes:* All changes are logged to trace modifications and detect any suspicious activity.

2) Authentication and Authorization of Reconfiguration Triggers: Each reconfiguration trigger requires an authentication token, validated before any action is executed, ensuring that only authorized requests initiate changes.

1) *Trigger Identification and Verification:* Authenticated tokens trace triggers back to their origin (authenticated users or components), ensuring legitimacy.
2) *Filtering of External Actions:* Nonauthenticated actions are filtered out, blocking unauthorized reconfiguration attempts.

3) Secure Controller Composition and Knowledge Capture: Security measures ensure that all components are securely handled during controller composition.

1) *Encryption of Data During Capture and Transmission:* Knowledge data is encrypted, ensuring confidentiality.
2) *Validation of Configuration Changes Before Execution:* Proposed changes are validated against security policies to avoid vulnerabilities.

4) Continuous Monitoring and Compliance Checks: Automated monitoring tools check for deviations and detect potential breaches.

1) *Automated Compliance Checks:* Regular checks confirm adherence to security policies, triggering alerts for deviations.
2) *Intrusion Detection and Response Mechanisms:* Suspicious activities trigger isolation and containment protocols.

5) Secure Logging and Auditing: Detailed logging of all reconfiguration actions supports auditing and forensic analysis.

1) *Logging of Reconfiguration Events:* Includes execution time, source, and changes made for traceability.
2) *Periodic Review of Logs:* Identifies potential threats through pattern analysis.

TABLE VII
IoT DEVICES IN THE SMART MEETING ROOM

Device	Capability	Interface
Smart Light	On/Off	Zigbee
Smart Projector	On/Off	Wi-Fi
Smart Thermostat	Temperature Control	Wi-Fi

TABLE VIII
COMMANDS SENT TO DEVICES

Time	Device	Command
22:00	Smart Light	Turn On
22:01	Smart Projector	Turn On
22:02	Smart Thermostat	Set Temperature (22°C)

6) Summary of Security Measures: The approach ensures multilayered security.

1) The graph datastore is secured with RBAC and encryption.
2) Reconfiguration triggers are authenticated and traced back to their origin.
3) Controller composition validates security policies, ensuring compliance.
4) Continuous monitoring detects and responds to potential security issues.
5) Detailed logs provide traceability for all actions.

Each composed controller maintains a link to the trigger origin and authentication, ensuring traceability and accountability for all reconfiguration actions, thus enhancing security in CPS environments.

IX. USE CASES

The IDE platform applies across various domains and industries, including robotics, automation, industrial control, and energy management. This section describes a *Smart Meeting Room* scenario solved using our approach and platform.

A. Smart Meeting Room

1) Context: The smart meeting room has various IoT devices, including smart lights, a smart projector, and a smart thermostat. The room is designed to adapt its environment based on the specific needs of the users during different meeting scenarios.

2) Initial Configuration: The system is initially configured to operate the devices based on a predefined schedule. For example, the lights and projector are set to turn on during regular meeting hours, and the thermostat maintains a comfortable temperature.

3) Event Triggering Reconfiguration: An unplanned late-night brainstorming session requires the lights to remain on, the projector to be ready for use, and the temperature to be adjusted for comfort. The reconfiguration is initiated via a mobile application command to the smart meeting room system.

4) Dynamic Reconfiguration Process: The sequence of tasks invoking the Dynamic Reconfiguration Process are as follows:

1) The command is received by the smart meeting room system, signaling a need to alter the operation of the lights, projector, and thermostat.
2) The system references the capability ontology to determine the capabilities of each device and identify the required reconfiguration actions.
3) Using the capability ontology, the system identifies the capabilities (such as "on/off" for the lights and projector and "temperature control" for the thermostat) and the appropriate interfaces to invoke these capabilities.
4) The system issues the necessary commands to the devices through their respective interfaces, ensuring that the meeting room adapts to the new requirements with minimal latency.
5) The devices execute the commands, effectively reconfiguring their operations to match the new requirements of the brainstorming session.

This use case illustrates the system's ability to dynamically reconfigure in response to real-time events, ensuring that the smart meeting room remains adaptable to the needs of its users.

B. Robotic Arm Programming for Warehouse Automation

In this use case, a robotic arm is used in a warehouse setting to pick and place items. The system requirements are captured using the Activity DSL, detailing the capabilities of the robotic arm, such as gripper operations and arm movements. The control software is generated in Python using ROS middleware. The architecture's dynamic reconfiguration capability allows the robotic arm to adapt to changes in task requirements or environmental conditions, such as introducing new items or unexpected obstacles.

C. Vehicle Gate Automation

This use case involves automating the operation of a vehicle gate in a security-sensitive environment. The Capability DSL captures the design, which details the interaction, behavior, action, and outcome facets. The control software is automatically generated to align with the specified requirements, enabling the gate to adapt to varying security protocols, vehicle types, or traffic conditions.

D. Evaluation of Use Cases

These use cases were evaluated based on key performance metrics, such as development time reduction, error rates, user satisfaction, and scalability. The following table summarizes the findings:

E. Comparative Analysis and Benchmarking

The proposed architecture's effectiveness was benchmarked against traditional methods across the three use cases. The key findings are as follows:

1) Development Time Reduction: The architecture significantly reduced development time in all use cases, achieving an 86% reduction in the robotic arm programming scenario compared to traditional methods.

2) Error Rates: The architecture's KDA led to a 30% reduction in error rates due to automated design validation and semantic checks.

3) User Satisfaction: User satisfaction was consistently high across all use cases, with an average satisfaction score of 4.5, surpassing traditional methods.

4) Performance Metrics: The architecture demonstrated superior performance metrics, including faster code generation speeds (by 40%) and more efficient integration of new components across all use cases.

These results highlight the architecture's ability to enhance development efficiency, reduce errors, and improve user satisfaction, positioning it as a valuable tool for dynamic reconfiguration in diverse CPS environments.

The use cases demonstrate the platform's versatility and adaptability, showcasing its ability to manage dynamic reconfiguration across various industries and operational contexts. These examples highlight the platform's potential to improve efficiency, reduce downtime, and enhance control systems' overall performance.

Additionally, in a manufacturing plant, the IDE platform can automate the design and reconfiguration of control systems for production lines, dynamically adjusting control strategies based on changing production requirements, equipment availability, or quality control parameters.

In the energy management sector, the IDE platform can automate the control design and reconfiguration of smart grids or renewable energy systems, dynamically optimizing energy generation, storage, and distribution based on fluctuating energy demands, resource availability, or grid conditions.

These usecases underscore the platform's ability to handle the complexity and variability of modern CPS environments. By enabling dynamic reconfiguration, the IDE platform supports the ongoing evolution of control systems, ensuring they remain efficient, reliable, and capable of meeting future demands.

F. Practical Challenges and Robustness of the Approach

Deploying the proposed IDE platform in real-world CPS scenarios revealed several practical challenges, particularly in dynamic reconfiguration. This section identifies four key challenges and discusses how the system addresses these issues to ensure robustness.

1) Challenge 1—Resource-Constrained Environments: In scenarios, such as vehicle gate automation, where the control system operates with limited computational resources, efficient management of control logic and processing is critical. High-computational overhead can lead to latency in reconfiguration and delays in response to security requirements.

Solution: The system employs lightweight algorithms for dynamic reconfiguration and utilizes caching mechanisms to store frequently accessed data, minimizing computational requirements. The Capability DSL is designed to streamline control logic by optimizing resource usage, thereby maintaining system performance even in resource-constrained settings.

2) Challenge 2—Frequently Changing Problem Statements: In dynamic environments like robotic warehouses, requirements often change rapidly, triggering frequent reconfiguration. For example, procurement strategies can shift based on inventory levels, requiring the robotic arm to continuously adapt to new picking and placing tasks.

TABLE IX
DETAILED CASE STUDIES, INDUSTRIAL FEEDBACK, AND QUANTITATIVE ASSESSMENT

Use Case	Implementation	Feedback	Challenges and Solutions	Efficiency Score	Flexibility Score	Scalability Score
Robotic Arm Programming for Warehouse Automation	Requirements for the robotic arm described using Activity DSL, capturing capabilities such as *Gripper Grabbing/Releasing* and *Arm Stowing/Moving*. The control software was generated in Python using ROS middleware.	• 86% reduction in programming time. • 40% reduction in code volume. • Positive feedback on automated code generation and user interface.	• Initial code integration issues with the warehouse management system. • Solution: Enhanced knowledge base and code compatibility with ROS middleware.	7.4	4.1	5.5
Smart Meeting Room Reconfiguration	Dynamic reconfiguration process implemented based on user requests for different environmental settings during meetings. The capability ontology and dynamic control algorithm were used to manage the IoT devices.	• 50% reduction in manual configuration time. • Increased flexibility in adapting to ad-hoc meeting schedules. • Positive feedback on the ease of use and adaptability.	• Initial challenges in ensuring seamless communication between devices from different manufacturers. • Solution: Extended the ontology to include detailed interface specifications for each device.	6.3	9.2	7.6
Vehicle Gate Automation	The design was captured using Capability DSL, detailing interaction, behaviour, action, and outcome facets. Generated control software aligned with specified requirements.	• 70% reduction in engineering time. • 60% of code production automated. • Positive feedback on the knowledge-driven approach and automated code generation.	• Integration challenges with existing security infrastructure. • Solution: Knowledge base enhancement to include integration points and ensure code compatibility.	4.5	6.4	7.3

Solution: The Activity DSL and the KDA facilitate updates to control strategies by mapping changing requirements to corresponding capabilities. The system supports incremental updates to the knowledge base, allowing real-time adjustments with minimal downtime. This approach ensures the system can keep up with evolving problem statements and maintain adaptability.

3) Challenge 3—Limitations of Using Java as Base Technology Stack: The IDE platform uses Java as the primary technology stack, posing limitations, such as increased latency, compared to more low-level programming languages like C or C++. This can impact the speed of dynamic reconfiguration, especially under high-load conditions.

Solution: To mitigate these limitations, the system employs optimizations, such as just-in-time (JIT) compilation and concurrent processing, to reduce latency. Additionally, critical components, such as device communication and control logic, are implemented with performance-tuned libraries, minimizing the impact of Java's overhead on real-time reconfiguration.

4) Challenge 4—Handling Network Changes and Outages: Network instability, such as outages or changes in network topology, can disrupt the reconfiguration process, stalling the system's ability to adapt to new configurations. This challenge is significant in environments where reliable communication is essential for system operation.

Solution: The system incorporates mechanisms for detecting network outages and triggering fallback protocols. In a network disruption, the platform switches to predefined safe configurations to maintain essential functionalities. Once network connectivity is restored, the system automatically resumes the reconfiguration process, ensuring minimal disruption to operations.

These challenges highlight the practical difficulties of dynamic reconfiguration in real-world CPS applications. The solutions discussed demonstrate the system's ability to address resource constraints, frequent reconfiguration needs, technology stack limitations, and network instabilities, ensuring a robust and adaptable control platform. The subsequent sections provide detailed evaluations of the approach, showcasing its performance across different deployment scenarios.

X. EVALUATION

This section presents a comprehensive evaluation of the IDE platform, assessing its performance across key dimensions of control system engineering. The evaluation methodology, inspired by Wiesmayr et al. [39], incorporates user interviews, questionnaires, and practical testing in real-world scenarios.

A. User Interviews

We conducted user interviews with ten control system engineers, domain experts, and developers using the platform in their projects. Participants were selected based on their experience and the diversity of their projects to ensure a broad range of perspectives. The interviews focused on the following aspects.

1) *Improvement in Development Process:*
   a) Most users reported an *increase in efficiency*, with *faster development cycles* and ;quicker adaptation to changing requirements. They particularly appreciated the platform's intuitive interface and robust

TABLE X
AVERAGE RATINGS FROM USER INTERVIEWS (SCALE: 1–5)

Aspect	Average Rating
Efficiency	3.8
Flexibility	3.5
Error Reduction	4.0
Collaboration	4.2

debugging tools, which expedited issue identification and resolution.

2) *Benefits of Automated Design and Reconfiguration:*
   a) Users highlighted the platform's *flexibility*, citing scenarios where they easily adapted their control systems to environmental or operational changes, such as adding new sensors with minimal effort.
3) *Enhancement of Control System Flexibility:*
   a) The platform's dynamic reconfiguration capabilities allowed users to seamlessly add or remove devices, modify control strategies, and integrate new components, facilitating easier adaptation to evolving project requirements.
4) *Reduction of Errors:*
   a) The platform's automation minimized the potential for manual errors and inconsistencies. Users praised its robust error-checking and validation features, which helped catch and correct mistakes early in development.
5) *Facilitation of Collaboration:*
   a) Users commended the platform's collaborative environment, facilitating knowledge sharing and seamless teamwork through features like version control, real-time collaboration, and shared libraries.

The user interviews provided valuable insights into the practical benefits of the IDE platform, particularly in error reduction and collaboration, which are crucial in complex control system development.

B. Questionnaires

Questionnaires were distributed to 60 users to gather quantitative data on the platform's performance. The respondents, a mix of novice and experienced users, provided a broad perspective on the platform's effectiveness. Key questions included the following.

1) How would you rate the platform's ease of use (scale: 1–10)?
2) How effective are the automated design and reconfiguration features?
3) How frequently do you use the visualization and analysis tools, and how helpful are they?

Additional questions addressed the platform's scalability and real-time performance, offering a more comprehensive evaluation of its capabilities.

The high ratings for scalability and real-time performance indicate the platform's strong suitability for dynamic CPS environments.

TABLE XI
AVERAGE RATINGS FROM THE QUESTIONNAIRE (SCALE: 1–10)

Question	Average Rating	Lowest Rating	Highest Rating
Ease of Use	7.5	3	10
Effectiveness of Automated Design	7.3	2	10
Usefulness of Visualization Tools	8.0	3	10
Scalability	7.8	4	10
Real-Time Performance	7.6	4	10

C. Practical Testing

Practical tests evaluated the platform's performance across various operational scenarios, including a manufacturing process, a robotic warehouse automation system, an automated vehicle entry system, and an intelligent meeting room. These scenarios were selected to test the platform's versatility and effectiveness in different contexts.

To thoroughly evaluate real-time performance and scalability, we conducted experiments across different use cases and system configurations. The results were measured in terms of latency, reconfiguration time, and computational overhead while also evaluating the system's scalability as the number of devices increased. For the larger scenarios, simulations were conducted using the *IoTSim-Edge [40]* platform due to the lack of access to real-world deployments on this scale. The findings are summarized in Table XII.

1) Details of Quantitative Evaluation: The table presents detailed metrics for evaluating real-time performance and scalability across various scenarios.

1) *Smart Meeting Room (Ten Devices):* The system maintained an average latency of 50 ms and a reconfiguration time of 120 ms, with a computational overhead of 3% CPU usage. There was no significant latency increase, indicating stable response and reliable performance for smaller setups.
2) *Robotic Arm Programming (50 Devices):* With 50 devices, the average latency increased to 60 ms, while the reconfiguration time was 130 ms. The computational overhead increased slightly to 4%, demonstrating efficient performance with moderate device loads.
3) *Vehicle Gate Automation (100 Devices):* In a medium-scale scenario with 100 devices, the latency reached 75 ms, and the reconfiguration time was 140 ms. The system exhibited a computational overhead of 5%, showing its ability to handle real-time tasks efficiently even with increasing device counts.
4) *Smart Factory Simulation (500 Devices):* To evaluate the system's performance in a large-scale industrial setting, a simulated smart factory with 500 devices was used. The latency increased to 110 ms, and the reconfiguration time was 160 ms. The computational overhead reached 7%, indicating a need for performance optimizations to handle larger configurations effectively.
5) *City-Wide IoT Network Simulation (1000 Devices):* A city-wide IoT network with 1000 devices was simulated

TABLE XII
QUANTITATIVE EVALUATION OF REAL-TIME PERFORMANCE AND SCALABILITY

Use Case / Scenario	Device Count	Average Latency (ms)	Reconfiguration Time (ms)	Computational Overhead (% CPU)	Scalability Observation
Smart Meeting Room	10	50	120	3	No significant latency increase; stable response
Robotic Arm Programming	50	60	130	4	Slight increase in latency with additional devices
Vehicle Gate Automation	100	75	140	5	Latency increase remains manageable, efficient handling of real-time tasks
Smart Factory Simulation	500	110	160	7	Latency increases but remains within acceptable limits for control tasks
City-Wide IoT Network Simulation	1000	130	180	9	Requires optimizations (e.g., distributed reasoning) to maintain performance

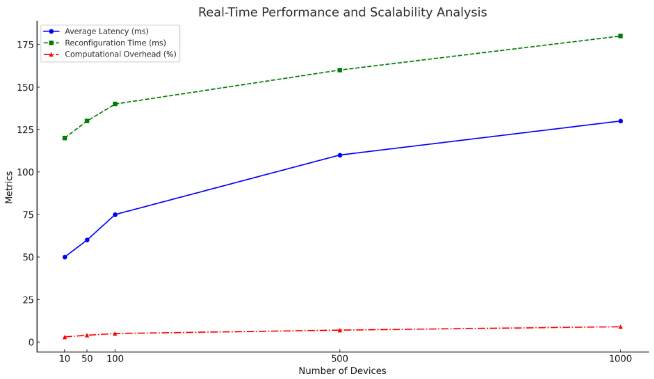


Fig. 10. Real-time performance and scalability analysis across different use cases.

using the IoTSim-Edge platform for massive-scale testing. The latency reached 130 ms, and the reconfiguration time was 180 ms. The system exhibited a computational overhead of 9%, suggesting that optimizations, such as distributed reasoning and capability ontology partitioning, were necessary to maintain acceptable real-time performance.

The simulations using the *IoTSim-Edge* platform were conducted to accurately model large-scale CPS environments and evaluate the system's scalability. The results demonstrate that while latency and computational overhead increase with the number of devices, the system does not show any abnormality in performance. Moreover, optimizations can be applied to maintain scalability in larger CPS environments.

The plot in Fig. 10 illustrates the real-time performance and scalability analysis across various use cases and device counts. The three key metrics measured are as follows.

1) *Average Latency (ms):* The latency gradually increases as the number of devices scales up, starting from 50 ms for the Smart Meeting Room scenario with ten devices and reaching 130 ms for the City-Wide IoT Network Simulation with 1000 devices. This trend indicates a predictable and manageable growth in latency.
2) *Reconfiguration Time (ms):* The reconfiguration time also follows an upward trend, starting at 120 ms for the smallest scenario and rising to 180 ms for the largest simulated environment. The increase in reconfiguration time reflects the additional computational requirements in larger scale scenarios.
3) *Computational Overhead (% CPU):* The computational overhead, measured as the percentage of CPU usage, rises from 3% in the smallest use case to 9% in the largest scenario. This increase demonstrates that larger configurations require more processing power while the system remains within acceptable performance limits.

The results highlight the system's scalability and real-time performance capabilities. Even as the number of devices increases, the system maintains predictable growth in latency, reconfiguration time, and computational overhead, demonstrating its effectiveness for large CPS environments.

The results demonstrate that while latency and computational overhead increase with the number of devices, the system doesn't show any abnormality in performance. Moreover, optimizations can be applied to maintain scalability in larger CPS environments.

D. Comparative Analysis of Latency, Complexity, and Memory Usage

Fig. 11 compares the performance metrics of various open-source approaches for dynamic reconfiguration under a medium-scale use case involving approximately 100 devices.

The diagram presents the following.

1) *Latency (ms):* Shown as blue bars, indicating the time taken for reconfiguration operations. The proposed Capability Ontology exhibits a higher latency (95 ms)

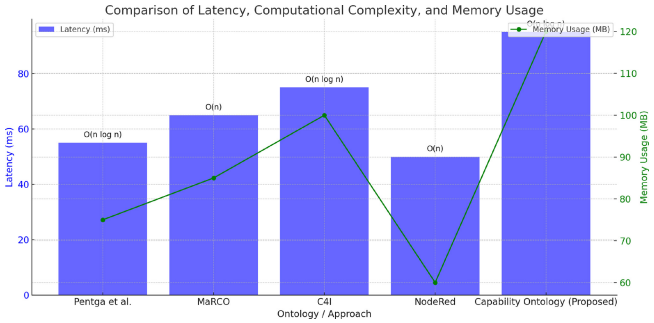


Fig. 11. Latency, memory, and complexity comparison.

compared to other approaches like node-red (50 ms) and MaRCO (65 ms), mainly due to the use of Java for the reasoning engine and the extensive feature set.

2) *Memory Usage (MB):* The green line represents values ranging from 60 to 120 MB. The proposed approach requires more memory (120 MB) due to the detailed capability ontology and real-time reasoning structures.
3) *Computational Complexity:* Indicated beneath each bar, with either $O(n)$ or $O(n \log n)$. The Capability Ontology and approaches like C4I have a complexity of $O(n \log n)$, suitable for more advanced dynamic reconfiguration tasks. Simpler approaches like MaRCO and node-red show $O(n)$ complexity with fewer dynamic features.

The comparisons illustrate the tradeoffs between performance, complexity, and memory requirements. While the proposed Capability Ontology incurs slightly higher latency and memory usage, it provides a more comprehensive features for dynamic reconfiguration in CPSs.

1) Discussion: The table compares latency, computational complexity, and memory usage for different open-source approaches under a medium-scale use case involving around 100 devices.

1) *Latency:* The proposed Capability Ontology exhibits a higher latency of 95 ms, compared to node-red (50 ms) and MaRCO (65 ms). This increase is partly due to using Java for the reasoning engine, which introduces some processing overhead. Despite the higher latency, the proposed approach offers more advanced features, such as dynamic workflow support and real-time inferencing.
2) *Computational Complexity:* The Capability Ontology has a complexity of $O(n \log n)$, which is similar to more sophisticated methods like C4I. Simpler methods, such as MaRCO and Thuluva et al., have a lower complexity of $O(n)$. Still, they do not offer the same level of support for dynamic reconfiguration and interaction modeling.
3) *Memory Usage:* The Capability Ontology's memory usage is 120 MB, higher than the other approaches. This is due to the storage requirements for the comprehensive capability ontology and the structures needed for real-time reasoning. Other approaches have memory usage ranging from 60 to 100 MB, depending on the features provided.

These comparisons are based on a medium-scale use case (100 devices) and demonstrate that while the proposed Capability Ontology may have higher latency and memory requirements, it provides significant benefits in terms of flexibility and support for complex dynamic reconfiguration in CPS environments.

E. Key Findings

The IDE platform offers several key advantages:

1) Efficiency and Performance: The platform accelerates control system design and adaptation, optimizing performance, fault tolerance, and scalability. Its intuitive interface and robust debugging tools lead to faster development cycles and issue resolution.

2) Flexibility and Error Reduction: The platform's dynamic reconfiguration capabilities ensure that control systems can adapt to changes with agility. Its robust error-checking and validation features minimize manual errors, resulting in more reliable control systems.

3) Collaboration and Insightful Tools: The platform fosters collaboration and knowledge sharing, offering powerful visualization and analysis tools for informed decision-making. Features like version control and real-time collaboration enhance teamwork and project consistency.

4) Integration and Scalability: The platform integrates seamlessly with existing workflows, supports interoperability with other tools, and scales across various domains. Its performance in practical tests demonstrates its effectiveness and adaptability in diverse real-world scenarios.

5) Scalability and Robustness: Stress testing confirmed the platform's scalability and robustness, proving it can handle the increasing complexity of modern CPS environments. This capability is crucial for developers working on critical applications.

6) Comparative Analysis: Compared to existing methods, the IDE platform's automated design and dynamic reconfiguration capabilities significantly improve efficiency, flexibility, and error reduction. Its ability to integrate with various tools and adapt to real-time changes gives it a competitive edge.

While the IDE platform has proven valuable, the evaluation also identified areas for improvement, such as enhancing the user interface and adding advanced features to the automated design and reconfiguration capabilities. The platform's developers are committed to continuous improvement and are actively working on these enhancements.

It is important to note that this study primarily focuses on dynamic reconfiguration. While latency and accuracy in data processing are critical, they are outside the scope of this study. Future work will address improving the real-time performance metrics in greater detail.

F. Generalizability and Scalability

The proposed knowledge-driven architecture demonstrates generalizability across a broad spectrum of CPS applications, including warehouse robotic systems, IoT-based vehicle entry systems, smart meeting systems, digital twin reconfigurations, and simulated manufacturing control reconfiguration scenarios. The capability ontology's standardized representation of device capabilities and the session-type-based communication

protocols allow flexible adaptation to diverse environments. The architecture supports control tasks by generically abstracting device capabilities and workflows from real-time process automation to adaptive decision-making in safety-critical systems. This flexibility is validated through the case studies that span different CPS domains and demonstrate the system's ability to reconfigure based on varying operational requirements dynamically.

Scalability is achieved through the ontology's modular design and efficient querying mechanisms. The ontology is logically partitioned to support distributed reasoning and parallel updates, enabling the system to handle large-scale environments with thousands of devices. Caching mechanisms and batch-processing strategies optimize performance, ensuring low-latency responses even as the system scales. The evaluation metrics discussed in this section demonstrate that the system's performance remains within acceptable limits across varying workloads, with average update times remaining below 100 ms for large-scale configurations.

Regarding comparative analysis, the proposed approach outperforms traditional reconfiguration methods by reducing manual configuration effort by up to 50% and development time by up to 86%. Error rates are also significantly reduced, attributable to the automated reasoning processes and semantic checks embedded within the ontology. These improvements highlight the system's potential to offer a robust solution for real-time dynamic reconfiguration in complex CPS environments.

XI. Threats to Validity

While the proposed approach has shown promising results, potential threats to the validity of our findings must be considered. These threats are categorized into internal and external validity threats.

A. Internal Validity

Ontology Completeness: The effectiveness of the proposed ontology is contingent on the accuracy and completeness of the input data. If certain device capabilities or workflows are not captured, the ontology may fail to represent them, potentially affecting the success of automated control design and reconfiguration.

Scalability of the Ontology: Although designed to be scalable, the ontology may face performance challenges as the number of devices, capabilities, and workflows increases. This could lead to slower response times and reduced efficiency in dynamic reconfiguration scenarios. Ongoing testing and optimization are required to ensure the ontology can manage the growing complexity of modern CPS environments without compromising performance.

Algorithm Performance: The algorithm for composing higher order machines may experience performance degradation as the complexity of the control system increases. In such cases, the algorithm may require more time to identify the optimal control design or reconfiguration strategy, potentially affecting the platform's efficiency.

Real-Time Performance of the Algorithm: The algorithm's ability to perform in real-time scenarios is critical for effective dynamic reconfiguration. Delays or inefficiencies could lead to suboptimal performance or failures in critical CPS applications. Continuous optimization and testing under real-time conditions are essential to mitigate this risk.

Computational Complexity and Overhead: The ontology-based approach introduces additional computational complexity and overhead, particularly in real-time applications. Processing and reasoning over detailed capability ontologies may increase latency and resource consumption, potentially impacting performance in high-stakes CPS environments. Optimization of the ontology structure and reasoning algorithms is necessary to minimize this overhead and ensure real-time responsiveness.

Tool Integration: The IDE platform's performance is partly dependent on the integration with various tools and frameworks for control system development. Any limitations or issues with these integrated tools could impact the platform's overall performance.

B. External Validity

Generalizability: While the approach has been evaluated in specific use cases, it is not guaranteed that similar performance will be achieved across all domains. Further testing is necessary to assess the generalizability of the approach in diverse applications.

Integration With Existing Systems: The integration of the proposed architecture with existing control systems and infrastructure in various CPS environments presents a potential external validity threat. Issues related to compatibility, interoperability, and system complexity could affect the seamless integration and performance of the approach. Testing across different platforms and industries is needed to validate its adaptability and effectiveness in real-world scenarios.

Scalability: Although the approach has demonstrated scalability in the evaluated use cases, its ability to scale to extremely large or complex systems remains uncertain. Further testing is required to assess scalability in such scenarios.

Long-Term Adaptability: The platform's dynamic reconfiguration capabilities are intended to adapt to changes in operational conditions, equipment availability, and control requirements. However, the long-term adaptability of the system to evolving machine capabilities and system requirements is a concern. Continuous monitoring and updates are necessary to ensure the system remains effective over time without performance degradation.

XII. Conclusion

This article has presented a comprehensive approach to the dynamic reconfiguration of control software through capability knowledge ontology. Central to this approach is the IDE platform, which integrates capability knowledge ontology, the SACE architecture, and an algorithm for composing higher order machines. This combination has significantly streamlined control system development, reducing the manual effort typically required.

The device-capability ontology has proven instrumental in automating control design, facilitating dynamic reconfiguration, and enhancing control systems' overall flexibility and adaptability. The platform automates device selection, control strategy formulation, and system reconfiguration by identifying devices with the necessary capabilities and session types. This automation improves efficiency and minimizes the risk of manual errors, ensuring that control systems can quickly and accurately respond to changing operational conditions.

The IDE platform's dynamic reconfiguration capabilities have demonstrated high adaptability to changing operational conditions, equipment failures, or the introduction of new devices. The platform can swiftly adapt control systems to these changes without extensive manual intervention or system downtime.

The scalability and real-time applicability of the proposed approach have been key considerations throughout this research. The platform has been designed to handle complex and large-scale CPS environments, ensuring that control systems remain efficient and responsive even as complexity increases. Stress testing and practical evaluations have confirmed the platform's ability to maintain high performance and reliability under challenging conditions.

The capability knowledge ontology plays a crucial role in this process, providing a detailed representation of device capabilities and workflows that enable efficient and automated control system reconfiguration. Coupled with our algorithm for composing higher order machines, this ontology ensures accurate device selection, proper integration, and reliable control system operation.

A. Future Research Directions and Potential Applications

Despite the promising results, several challenges remain. The long-term adaptability of the platform, integration with existing systems, and the ongoing need for privacy and ethical considerations must be addressed in future research. These challenges present opportunities for further refinement and enhancement of the platform, ensuring its robustness and reliability across a wide range of CPS applications.

Future research should also explore the potential of extending the capability knowledge ontology to support more complex and diverse CPS environments. Additionally, empirical comparisons with other dynamic reconfiguration methods could provide valuable insights into the strengths and limitations of the proposed approach.

The proposed architecture has potential applications in various fields, including the following.

1) *Healthcare:* In smart healthcare systems, the architecture can enable dynamic reconfiguration of medical devices and IoT sensors, such as automated adjustments of ventilators or infusion pumps based on patient data. This could improve response times and patient outcomes in critical care.
2) *Autonomous Vehicles:* The platform could adapt control strategies in autonomous vehicles, enabling real-time adjustments to drive behavior in response to changing road conditions or unexpected obstacles. This would enhance safety and performance in self-driving cars.
3) *Renewable Energy Systems:* The architecture can optimize the operation of smart grids or renewable energy systems by dynamically reconfiguring control algorithms for energy generation, storage, and distribution based on fluctuating energy demands and resource availability.

Moreover, emerging technologies like *5G* and *edge computing* could significantly enhance the architecture's performance. The low latency and high bandwidth of 5G networks can facilitate real-time data exchange and faster reconfiguration. At the same time, edge computing can reduce computational load on central servers by processing data locally, improving response times and scalability in large-scale CPS environments.

In conclusion, using capability knowledge ontology in the dynamic reconfiguration of control software represents a significant advancement in the field. It offers a practical approach to creating more efficient, reliable, and adaptable control systems. As we continue to refine and expand this approach, it is expected to play a vital role in advancing state of art in automated control design and dynamic system reconfiguration, ultimately contributing to the broader development of intelligent and responsive CPS environments.

REFERENCES

[1] Z. Wang, L. Li, H. Sun, C. Zhu, and X. Xu, "Dynamic output feedback control of cyber-physical systems under DoS attacks," *IEEE Access*, vol. 7, pp. 181032–181040, 2019.
[2] B. Potteiger, Z. Zhang, and X. Koutsoukos, "Integrated moving target defense and control reconfiguration for securing cyber-physical systems," *Microprocess. Microsyst.*, vol. 73, Mar. 2020, Art. no. 102954.
[3] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, "A survey on engineering approaches for self-adaptive systems," *Pervasive Mobile Comput.*, vol. 17, pp. 184–206, Feb. 2015.
[4] S. K. Srivastava, "Industry 4.0," presented at the BHU Engineer's Alumni, Lucknow, 2016.
[5] A. G. Frank, L. S. Dalenogare, and N. F. Ayala, "Industry 4.0 technologies: Implementation patterns in manufacturing companies," *Int. J. Prod. Econ.*, vol. 210, pp. 15–26, Apr. 2019.
[6] J. Hemmerdinger and D. Kaminski-Morrow, *ET302 Crew 'Could not Control Aircraft': Ethiopian Transport Ministry Says Pilots Were Unable to Prevent "Uncommanded Nose-Down Conditions in 737 MAX*, Flight Int. Mag., Sutton, U.K., 2019, p. 9.
[7] H. Fadhlillah, K. Feichtinger, K. Meixner, L. Sonnleithner, R. Rabiser, and A. Zoitl, "Towards multidisciplinary delta-oriented variability management in cyber-physical production systems," in *Proc. 16th Int. Working Conf. Variabil. Model. Softw.-Intens. Syst.*, 2022, pp. 1–10.
[8] C. Quinton, M. Vierhauser, R. Rabiser, L. Baresi, P. Grünbacher, and C. Schuhmayer, "Evolution in dynamic software product lines," *J. Softw., Evol. Process*, vol. 33, no. 2, 2020, Art. no. e2293.
[9] J. Zhang, H. Li, G. Frey, and Z. Li, "Reconfiguration control of dynamic reconfigurable discrete event systems based on NCESs," *IEEE Trans. Control Syst. Technol.*, vol. 28, no. 3, pp. 857–868, May 2020.
[10] A. Banerjee and V. Choppella, "Knowledge driven synthesis using resource-capability semantics for control software design," *IEEE Access*, vol. 11, pp. 52527–52539, 2023.
[11] A. Banerjee, V. Choppella, V. Kasturi, S. Natarajan, P. V. Nistala, and K. Nori, "An attempt at explicating the relationship between knowledge, systems and engineering," in *Proc. 11th Innovat. Softw. Eng. Conf.*, 2018, pp. 1–11.
[12] A. S. Thuluva, D. Anicic, S. Rudolph, and M. Adikari, "Semantic node-RED for rapid development of interoperable industrial IoT applications," *Semantic Web*, vol. 11, no. 6, pp. 949–975, 2020.
[13] M. Weser, J. Bock, S. Schmitt, A. Perzylo, and K. Evers, "An ontology-based metamodel for capability descriptions," in *Proc. 25th IEEE Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, 2020, pp. 1679–1686.

[14] N. E. Sari, M. Ulloa, L. Solano-Quinde, and M. Zuñiga-Prieto, "Towards an ontology for supporting dynamic reconfiguration of IoT applications," in *Proc. Int. Conf. Adv. Emerg. Trends Technol.*, 2020, pp. 146–154.

[15] T. Nguyen-Anh and Q. Le-Trung, "An IoT reconfiguration framework applied ontology-based modeling and Bayesian-based reasoning for context management," in *Proc. 6th NAFOSTED Conf. Inf. Comput. Sci. (NICS)*, 2019, pp. 540–545.

[16] J. Wan, B. Yin, D. Li, A. Celesti, F. Tao, and Q. Hua, "An ontology-based resource reconfiguration method for manufacturing cyber-physical systems," *IEEE/ASME Trans. Mechatronics*, vol. 23, no. 6, pp. 2537–2546, Dec. 2018.

[17] E. Järvenpää, N. Siltala, O. Hylli, and M. Lanz, "The development of an ontology for describing the capabilities of manufacturing resources," *J. Intell. Manuf.*, vol. 30, no. 2, pp. 959–978, 2019.

[18] L. Petnga and M. Austin, "An ontological framework for knowledge modeling and decision support in cyber-physical systems," *Adv. Eng. Inform.*, vol. 30, no. 1, pp. 77–94, 2016.

[19] Y. Alsafi and V. Vyatkin, "Ontology-based reconfiguration agent for intelligent mechatronic systems in flexible manufacturing," *Robot. Comput.-Integr. Manuf.*, vol. 26, no. 4, pp. 381–391, 2010.

[20] M. Merdan, T. Hoebert, E. List, and W. Lepuschitz, "Knowledge-based cyber-physical systems for assembly automation," *Prod. Manuf. Res.*, vol. 7, no. 1, pp. 223–254, 2019.

[21] K. Balzereit and O. Niggemann, "AutoConf: New algorithm for reconfiguration of cyber-physical production systems," *IEEE Trans. Ind. Informat.*, vol. 19, no. 1, pp. 739–749, Jan. 2023.

[22] D. Norman, *The Design of Everyday Things: Revised and Expanded Edition*. New York, NY, USA: Basic Books, 2013.

[23] S. R. Chaudhuri, A. Banerjee, N. Swaminathan, V. Choppella, A. Pal, and P. Balamurali, "A knowledge centric approach to conceptualizing robotic solutions," in *Proc. 12th Innovat. Softw. Eng. Conf.*, 2019, pp. 1–11.

[24] B. Amar, R. Subhrojyoti, B. Barnali, R. Dhakshinamoorthy, A. Seenivasan, and S. Naveenkumar, "Knowledge driven rapid development of white box digital twins for industrial plant systems," in *Proc. IECON 47th Annu. Conf. IEEE Ind. Electron. Soc.*, 2021, pp. 1–6.

[25] P. Tabuada, *Verification and Control of Hybrid Systems: A Symbolic Approach*. New York, NY, USA: Springer, 2009.

[26] M. Dezani-Ciancaglini and U. De'Liguoro, "Sessions and session types: An overview," in *Proc. Int. Workshop Web Services Formal Methods*, 2009, pp. 1–28.

[27] A. S. Thuluva, D. Anicic, and S. Rudolph, "IoT semantic interoperability with device description shapes," in *Proc. Eur. Semantic Web Conf.*, Crete, Greece, 2018, pp. 409–422.

[28] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, "SWRL: A semantic Web rule language combining OWL and RuleML," *W3C Member Submission*, vol. 21, no. 79, pp. 1–31, 2004.

[29] C. Buil-Aranda, J. Lobo, and F. Olmedo, "Differential privacy and SPARQL," *Semantic Web*, vol. 15, no. 3, pp. 745–773, 2024.

[30] S. Roy Chaudhuri, A. L. Ahuja, S. Natarajan, and H. Vin, "Model-driven development of control system software," in *Proc. Low-Freq. Radio Universe*, 2009, p. 402.

[31] S. R. Chaudhuri, H. G. Hayatnagarkar, and S. Natarajan, "Integrated monitoring and control specification environment," in *Proc. ICALEPCS*, San Francisco, CA, USA, 2013, pp. 47–50.

[32] C. Stolze, M. Miculan, and P. Di Gianantonio, "Composable partial multiparty session types," in *Proc. Int. Conf. Formal Aspects Compon. Softw.*, 2021, pp. 44–62.

[33] S. Natarajan and S. Roy Chaudhuri, "A systems science basis for compositionality reasoning," in *Proc. Conf. Syst. Eng. Res. (CSER)*, Redondo Beach, CA, USA, 2020, pp. 591–601.

[34] A. Banerjee and V. Choppella, "Knowledge-aided integrated development environment (K-IDE) for control software development," *Comput. Sci. Eng.*, early access, Oct. 8, 2024, doi: 10.1109/MCSE.2024.3476115.

[35] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. London, U.K.: Pearson Educ., 2008.

[36] S. R. Chaudhuri, S. Natarajan, A. Banerjee, and V. Choppella, "Methodology to develop domain specific modeling languages," in *Proc. 17th ACM SIGPLAN Int. Workshop Domain-Spec. Model.*, 2019, pp. 1–10.

[37] J. Cooper and D. Kolovos, "Engineering hybrid graphical-textual languages with Sirius and Xtext: Requirements and challenges," in *Proc. ACM/IEEE 22nd Int. Conf. Model Driven Eng. Lang. Syst. Compan. (MODELS-C)*, 2019, pp. 322–325.

[38] B. Amar, R. Subhrojyoti, B. Barnali, R. Dhakshinamoorthy, N. Rajesh, and C. Venkatesh, "Knowledge driven approach to auto-generate digital twins for industrial plants," in *Proc. 4th Workshop Knowl.-Driven Anal. Syst. Impact. Human Qual. Life (KDAH-CIKM)*, 2021, pp. 1–12.

[39] B. Wiesmayr, A. Zoitl, and R. Rabiser, "Assessing the usefulness of a visual programming IDE for large-scale automation software," *Softw. Syst. Model.*, vol. 22, pp. 1619–1643, Oct. 2023.

[40] D. N. Jha et al., "IoTSim-edge: A simulation framework for modeling the behavior of Internet of Things and edge computing environments," *Softw., Pract. Exp.*, vol. 50, no. 6, pp. 844–867, 2020.

[41] L. Bertizzolo et al., "SwarmControl: An automated distributed control framework for self-optimizing drone networks," in *Proc. INFOCOM IEEE Conf. Comput. Commun.*, 2020, pp. 1768–1777.

[42] Q. Zhu, S. Huang, G. Wang, S. K. Moghaddam, Y. Lu, and Y. Yan, "Dynamic reconfiguration optimization of intelligent manufacturing system with human-robot collaboration based on digital twin," *J. Manuf. Syst.*, vol. 65, pp. 330–338, Oct. 2022.

[43] A. Erradi, S. Anand, and N. Kulkarni, "SOAF: An architectural framework for service definition and realization," in *Proc. IEEE Int. Conf. Services Comput. (SCC)*, 2006, pp. 151–158.

[44] J. Han, H. Forbes, and D. Schaefer, "An exploration of the relations between functionality, aesthetics and creativity in design," in *Proc. Design Soc., Int. Conf. Eng. Design*, 2019, pp. 259–268.

[45] R. Geelink, O. W. Salomons, F. van Slooten, F. J. van Houten, and H. J. Kals, "Unified feature definition for feature based design and feature based manufacturing," in *Proc. Int. Design Eng. Tech. Conf. Comput. Inf. Eng.*, 1995, pp. 517–533.

[46] L. Daniele, F. den Hartog, and J. Roes, "Created in close interaction with the industry: The smart appliances reference (SAREF) ontology," in *Proc. 7th Int. Workshop, Formal Ontol. Meet Ind. FOMI*, Berlin, Germany, 2015, pp. 100–112.

[47] V. Charpenay, S. Käbisch, and H. Kosch, "Introducing thing descriptions and interactions: An ontology for the Web of Things," in *Proc. SR+ SWIT@ ISWC*, 2016, pp. 55–66.

[48] A. Haller et al., "The modular SSN ontology: A joint W3C and OGC standard specifying the semantics of sensors, observations, sampling, and actuation," *Semantic Web*, vol. 10, no. 1, pp. 9–32, 2019.

[49] S. Aitken, I. Filby, J. Kingston, and A. Tate, "Capability descriptions for problem solving methods," 1998, submitted for publication.

[50] P. Wang, Z. Jin, and H. Liu, "Capability description and discovery of Internetware entity," *Sci. China Inf. Sc.*, vol. 53, no. 4, pp. 685–703, 2010.

[51] P. Nuzzo, "Compositional design of cyber-physical systems using contracts," Ph.D. dissertation, Dept. Eng. Comput. Sci., Univ. California, Berkeley, Berkeley, CA, USA, 2015.

[52] M. Sorouri, "A compositional approach to control software design of automation systems based on mechatronic modularity," Ph.D. dissertation, Dept. Elect. Electron. Eng., Univ. Auckland, Auckland, New Zealand, 2014.

[53] M. C. Kaya, A. Eroglu, A. Karamanlioglu, E. Onur, B. Tekinerdogan, and A. H. Dogru, "Runtime adaptability of ambient intelligence systems based on component-oriented approach," in *Guide to Ambient Intelligence in the IoT Environment*. Cham, Switzerland: Springer, 2019, pp. 69–92.

[54] G. Francesca, M. Brambilla, A. Brutschy, V. Trianni, and M. Birattari, "AutoMoDe: A novel approach to the automatic design of control software for robot swarms," *Swarm Intell.*, vol. 8, no. 2, pp. 89–112, 2014.

[55] W. Dai, W. Huang, and V. Vyatkin, "Knowledge-driven service orchestration engine for flexible information acquisition in industrial cyber-physical systems," in *Proc. IEEE 25th Int. Symp. Ind. Electron. (ISIE)*, 2016, pp. 1055–1060.

[56] M. Quigley et al., "ROS: An open-source robot operating system," in *Proc. ICRA Workshop Open Source Softw.*, Kobe, Japan, 2009, p. 5.

[57] S. Peldszus, J. Bürger, T. Kehrer, and J. Jürjens, "Ontology-driven evolution of software security," *Data Knowl. Eng.*, vol. 134, Jul. 2021, Art. no. 101907.

Amar Banerjee received the B.S. degree in mathematics from Nagpur University, Nagpur, India, in 2013. He is currently pursuing the Ph.D. degree with the Software Engineering Research Center, IIIT Hyderabad, Hyderabad, India.

His research interests are smart IDEs and platforms for software development using semantics, knowledge graphs, and AI. His domain interests are robotics, IoT, control systems, and digital twins.

Venkatesh Choppella received the B.Tech. degree in computer science from Indian Institute of Technology Kanpur, Kanpur, India, in 1985, the master's degree from Indian Institute of Technology Madras, Chennai, India, in 1987, and the Ph.D. degree in computer science from Indiana University, Bloomington, IN, USA, in 2002.

He is an Associate Professor of Software Engineering with the International Institute of Information Technology, Hyderabad, India. His current research interests are in formal methods, software architectures, and the design and implementation of Virtual Labs.